A formally verified compiler back-end

Xavier Leroy

Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A formally verified compiler back-end

Xavier Leroy

Received: 21 July 2009 / Accepted: 22 October 2009

Abstract This article describes the development and formal verification (proof of semantic preservation) of a compiler back-end from Cminor (a simple imperative intermediate language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its soundness. Such a verified compiler is useful in the context of formal methods applied to the certification of critical software: the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Keywords Compiler verification · semantic preservation · program proof · formal methods · compiler transformations and optimizations · the Coq theorem prover

1 Introduction

Can you trust your compiler? Compilers are generally assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers—and especially optimizing compilers—are complex programs that perform complicated symbolic transformations. Despite intensive testing, bugs in compilers do occur, causing the compiler to crash at compile time or—much worse—to silently generate an incorrect executable for a correct source program [67,65,31].

For low-assurance software, validated only by testing, the impact of compiler bugs is low: what is tested is the executable code produced by the compiler; rigorous testing should expose compiler-introduced errors along with errors already present in the source program. Note, however, that compiler-introduced bugs are notoriously difficult to track down. Moreover, test plans need to be made more complex if optimizations are to be tested: for example, loop unrolling introduces additional limit conditions that are not apparent in the source loop.

The picture changes dramatically for safety-critical, high-assurance software. Here, validation by testing reaches its limits and needs to be complemented or even replaced by the use of formal methods: model checking, static analysis, program proof, etc..

X. Leroy
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
E-mail: Xavier.Leroy@inria.fr

Almost universally, formal methods are applied to the source code of a program. Bugs in the compiler that is used to turn this formally verified source code into an executable can potentially invalidate all the guarantees so painfully obtained by the use of formal methods. In a future where formal methods are routinely applied to source programs, the compiler could appear as a weak link in the chain that goes from specifications to executables.

The safety-critical software industry is aware of these issues and uses a variety of techniques to alleviate them: even more testing (of the compiler and of the generated executable); turning compiler optimizations off; and in extreme cases, conducting manual code reviews of the generated assembly code. These techniques do not fully address the issue and are costly in terms of development time and program performance.

An obviously better approach is to apply formal methods to the compiler itself in order to gain assurance that it preserves the semantics of the source programs. Many different approaches have been proposed and investigated, including on-paper and on-machine proofs of semantic preservation, proof-carrying code, credible compilation, translation validation, and type-preserving compilers. (These approaches are compared in section 2.)

For the last four years, we have been working on the development of a *realistic*, *verified* compiler called Compcert. By *verified*, we mean a compiler that is accompanied by a machine-checked proof that the generated code behaves exactly as prescribed by the semantics of the source program (semantic preservation property). By *realistic*, we mean a compiler that could realistically be used in the context of production of critical software. Namely, it compiles a language commonly used for critical embedded software: not Java, not ML, not assembly code, but a large subset of the C language. It produces code for a processor commonly used in embedded systems, as opposed e.g. to a virtual machine: we chose the PowerPC because it is popular in avionics. Finally, the compiler must generate code that is efficient enough and compact enough to fit the requirements of critical embedded systems. This implies a multi-pass compiler that features good register allocation and some basic optimizations.

This paper reports on the completion of a large part of this program: the formal verification of a lightly-optimizing compiler back-end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor. This verification is mechanized using the Coq proof assistant [25, 11]. Another part of this program—the verification of a compiler front-end translating a subset of C called Clight down to Cminor—has also been completed and is described separately [15, 16].

While there exists a considerable body of earlier work on machine-checked correctness proofs of parts of compilers (see section 18 for a review), our work is novel in two ways. First, published work tends to focus on a few parts of a compiler, such as optimizations and the underlying static analyses [55, 19] or translation of a high-level language to virtual machine code [49]. In contrast, our work emphasizes end-to-end verification of a complete compilation chain from a structured imperative language down to assembly code through 6 intermediate languages. We found that many of the non-optimizing translations performed, while often considered obvious in compiler literature, are surprisingly tricky to prove correct formally.

Another novelty of this work is that most of the compiler is written directly in the Coq specification language, in a purely functional style. The executable compiler is obtained by automatic extraction of Caml code from this specification. This approach is an attractive alternative to writing the compiler in a conventional programming language, then using a program logic to relate it with its specifications. This approach

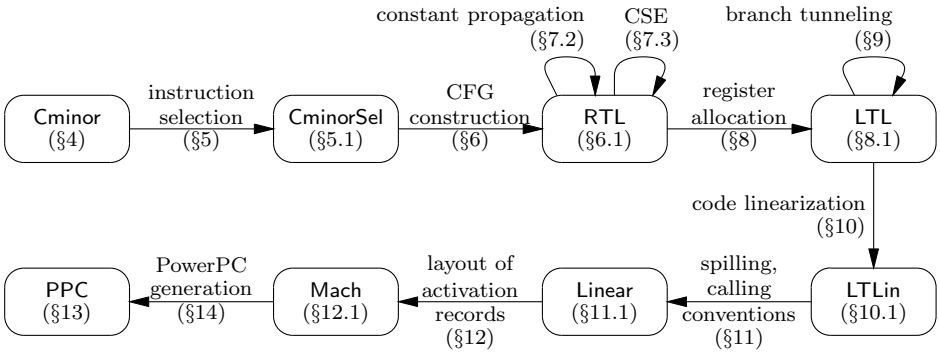

Fig. 1 The passes and intermediate languages of Compcert.

has never been applied before to a program of the size and complexity of an optimizing compiler.

The complete source code of the Coq development, extensively commented, is available on the Web [58]. We take advantage of this availability to omit proofs and a number of low-level details from this article, referring the interested reader to the Coq development instead. The purpose of this article is to give a high-level presentation of a verified back-end, with just enough details to enable readers to apply similar techniques in other contexts. The general perspective we adopt is to revisit classic compiler technology from the viewpoint of the semanticist, in particular by distinguishing clearly between the correctness-relevant and the performance-relevant aspects of compilation algorithms, which are inextricably mixed in compiler literature.

The remainder of this article is organized as follows. Section 2 formalizes various approaches to establishing trust in the results of compilation. Section 3 presents the main aspects of the development that are shared between all passes of the compiler: the value and memory models, labeled transition semantics, proofs by simulation diagrams. Sections 4 and 13 define the semantics of our source language Cminor and our target language PPC, respectively. The bulk of this article (sections 5 to 14) is devoted to the description of the successive passes of the compiler, the intermediate languages they operate on, and their soundness proofs. (Figure 1 summarizes the passes and the intermediate languages.) Experimental data on the Coq development and on the executable compiler extracted from it are presented in sections 15 and 16. Section 17 discusses some of the design choices and possible extensions. Related work is discussed in section 18, followed by concluding remarks in section 19.

2 General framework

2.1 Notions of semantic preservation

Consider a source program S and a compiled program C produced by a compiler. Our aim is to prove that the semantics of S was preserved during compilation. To make this notion of semantic preservation precise, we assume given semantics for the source language L_s and the target language L_t. These semantics associate one or several observable behaviors B to S and C. Typically, observable behaviors include termination,

divergence, and "going wrong" on executing an undefined computation. (In the remainder of this work, behaviors also contain traces of input-output operations performed during program execution.) We write $S \Downarrow B$ to mean that program S executes with observable behavior B, and likewise for C.

The strongest notion of semantic preservation during compilation is that the source program S and the compiled code C have exactly the same sets of observable behaviors—a standard bisimulation property:

Definition 1 (Bisimulation) $\forall B,\ S \Downarrow B \iff C \Downarrow B$.

Definition 1 is too strong to be usable as our notion of semantic preservation. If the source language is not deterministic, compilers are allowed to select one of the possible behaviors of the source program. (For instance, C compilers choose one particular evaluation order for expressions among the several orders allowed by the C specifications.) In this case, C will have fewer behaviors than S. To account for this degree of freedom, we can consider a backward simulation, or refinement, property:

Definition 2 (Backward simulation) $\forall B,\ C \Downarrow B \implies S \Downarrow B$.

Definitions 1 and 2 imply that if S always goes wrong, so does C. Several desirable optimizations violate this requirement. For instance, if S contains an integer division whose result is unused, and this division can cause S to go wrong because its second argument is zero, dead code elimination will result in a compiled program C that does not go wrong on this division. To leave more flexibility to the compiler, we can therefore restrict the backward simulation property to *safe* source programs. A program S is safe, written $\mathit{Safe}(S)$, if none of its possible behaviors is in the set `Wrong` of "going wrong" behaviors ($S \Downarrow B \implies B \notin \texttt{Wrong}$).

Definition 3 (Backward simulation for safe programs) If $\mathit{Safe}(S)$, then $\forall B,\ C \Downarrow B \implies S \Downarrow B$.

In other words, if S cannot go wrong (a fact that can be established by formal verification or static analysis of S), then neither does C; moreover, all observable behaviors of C are acceptable behaviors of S.

An alternative to backward simulation (definitions 2 and 3) is forward simulation properties, showing that all possible behaviors of the source program are also possible behaviors of the compiled program:

Definition 4 (Forward simulation) $\forall B,\ S \Downarrow B \implies C \Downarrow B$.

Definition 5 (Forward simulation for safe programs) $\forall B \notin \texttt{Wrong},\ S \Downarrow B \implies C \Downarrow B$.

In general, forward simulations are easier to prove than backward simulations (by structural induction on an execution of S), but less informative: even if forward simulation holds, the compiled code C could have additional, undesirable behaviors beyond those of S. However, this cannot happen if C is deterministic, that is, if it admits only one observable behavior ($C \Downarrow B_1 \wedge C \Downarrow B_2 \implies B_1 = B_2$). This is the case if the target language L_t has no internal non-determinism (programs change their behaviors only in response to different inputs but not because of internal choices) and the execution environment is deterministic (inputs given to programs are uniquely determined by their

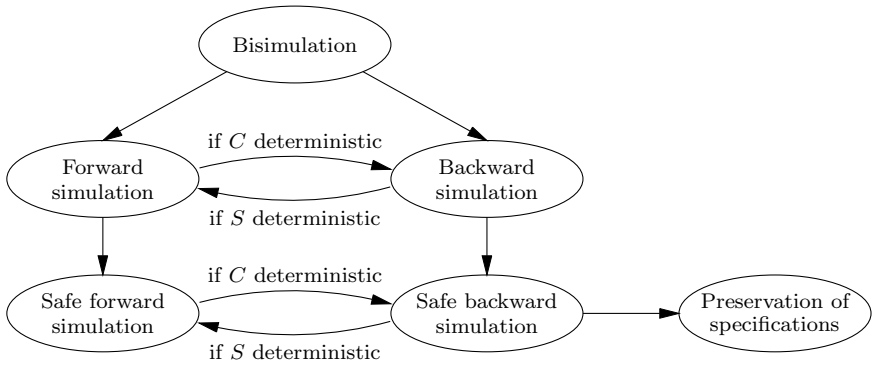

Fig. 2 Various semantic preservation properties and their relationships. An arrow from A to B means that A logically implies B.

previous outputs).[1] In this case, it is easy to show that "forward simulation" implies "backward simulation", and "forward simulation for safe programs" implies "backward simulation for safe programs". The reverse implications hold if the source program is deterministic. Figure 2 summarizes the logical implications between the various notions of semantic preservation.

From a formal methods perspective, what we are really interested in is whether the compiled code satisfies the functional specifications of the application. Assume that such a specification is given as a predicate $Spec(B)$ of the observable behavior. Further assume that the specification rules out "going wrong" behaviors: $Spec(B) \implies B \notin \texttt{Wrong}$. We say that C satisfies the specification, and write $C \models Spec$, if all behaviors of C satisfy $Spec$ ($\forall B,\ C \Downarrow B \implies Spec(B)$). The expected soundness property of the compiler is that it preserves the fact that the source code S satisfies the specification, a fact that has been established separately by formal verification of S.

Definition 6 (Preservation of a specification) $S \models Spec \implies C \models Spec$.

It is easy to show that "backward simulation for safe programs" implies "preservation of a specification" for all specifications *Spec*. In general, the latter property is weaker than the former property. For instance, if the specification of the application is "print a prime number", and S prints 7, and C prints 11, the specification is preserved but backward simulation does not hold. Therefore, definition 6 leaves more liberty for compiler optimizations that do not preserve semantics in general, but are correct for specific programs. However, it has the marked disadvantage of depending on the specifications of the application, so that changes in the latter can require the proof of preservation to be redone.

A special case of preservation of a specification, of considerable historical importance, is the preservation of type and memory safety, which we can summarize as "if S does not go wrong, neither does C":

Definition 7 (Preservation of safety) $Safe(S) \implies Safe(C)$.

[1] Section 13.3 formalizes this notion of deterministic execution environment by, in effect, restricting the set of behaviors B to those generated by a transition function that responds to the outputs of the program.

Combined with a separate check that S is well-typed in a sound type system, this property implies that C executes without memory violations. Type-preserving compilation [72, 71, 21] obtains this guarantee by different means: under the assumption that S is well typed, C is proved to be well-typed in a sound type system, ensuring that it cannot go wrong. Having proved a semantic preservation property such as definitions 3 or 6 provides the same guarantee without having to equip the target and intermediate languages with sound type systems and to prove type preservation for the compiler.

In summary, the approach we follow in this work is to prove a "forward simulation for safe programs" property (sections 5 to 14), and combine it with a separate proof of determinism for the target language (section 13.3), the latter proof being particularly easy since the target is a single-threaded assembly language. Combining these two proofs, we obtain that all specifications are preserved, in the sense of definition 6, which is the result that matters for users of the compiler who practice formal verification at the source level.

2.2 Verified compilers, validated compilers, and certifying compilers

We now discuss several approaches to establishing that a compiler preserves semantics of the compiled programs, in the sense of section 2.1. In the following, we write $S \approx C$, where S is a source program and C is compiled code, to denote one of the semantic preservation properties 1 to 7 of section 2.1.

2.2.1 Verified compilers

We model the compiler as a total function *Comp* from source programs to either compiled code (written $\mathit{Comp}(S) = \texttt{OK}(C)$) or a compile-time error (written $\mathit{Comp}(S) = \texttt{Error}$). Compile-time errors correspond to cases where the compiler is unable to produce code, for instance if the source program is incorrect (syntax error, type error, etc.), but also if it exceeds the capacities of the compiler (see section 12 for an example).

Definition 8 (Verified compiler) A compiler *Comp* is said to be verified if it is accompanied with a formal proof of the following property:

$$\forall S, C, \quad \mathit{Comp}(S) = \texttt{OK}(C) \Longrightarrow S \approx C \qquad (i)$$

In other words, a verified compiler either reports an error or produces code that satisfies the desired semantic preservation property. Notice that a compiler that always fails ($\mathit{Comp}(S) = \texttt{Error}$ for all S) is indeed verified, although useless. Whether the compiler succeeds to compile the source programs of interest is not a soundness issue, but a quality of implementation issue, which is addressed by non-formal methods such as testing. The important feature, from a formal methods standpoint, is that the compiler never silently produces incorrect code.

Verifying a compiler in the sense of definition 8 amounts to applying program proof technology to the compiler sources, using one of the properties defined in section 2 as the high-level specification of the compiler.

2.2.2 Translation validation with verified validators

In the translation validation approach [83, 76] the compiler does not need to be verified. Instead, the compiler is complemented by a *validator*: a boolean-valued function $Validate(S, C)$ that verifies the property $S \approx C$ *a posteriori*. If $Comp(S) = \texttt{OK}(C)$ and $Validate(S, C) = \texttt{true}$, the compiled code C is deemed trustworthy. Validation can be performed in several ways, ranging from symbolic interpretation and static analysis of S and C [76, 87, 44, 93, 94] to the generation of verification conditions followed by model checking or automatic theorem proving [83, 95, 4]. The property $S \approx C$ being undecidable in general, validators must err on the side of caution and should reply `false` if they cannot establish $S \approx C$.[2]

Translation validation generates additional confidence in the correctness of the compiled code but by itself does not provide formal guarantees as strong as those provided by a verified compiler: the validator could itself be unsound.

Definition 9 (Verified validator) A validator *Validate* is said to be verified if it is accompanied with a formal proof of the following property:

$$\forall S, C, \quad Validate(S, C) = \texttt{true} \Longrightarrow S \approx C \qquad (ii)$$

The combination of a verified validator *Validate* with an unverified compiler *Comp* does provide formal guarantees as strong as those provided by a verified compiler. Such a combination calls the validator after each run of the compiler, reporting a compile-time error if validation fails:

$$\begin{array}{l} Comp'(S) = \\ \quad \texttt{match } Comp(S) \texttt{ with} \\ \quad \mid \texttt{Error} \rightarrow \texttt{Error} \\ \quad \mid \texttt{OK}(C) \rightarrow \texttt{if } Validate(S, C) \texttt{ then OK}(C) \texttt{ else Error} \end{array}$$

If the source and target languages are identical, as is often the case for optimization passes, we also have the option to return the source code unchanged if validation fails, in effect turning off a potentially incorrect optimization:

$$\begin{array}{l} Comp''(S) = \\ \quad \texttt{match } Comp(S) \texttt{ with} \\ \quad \mid \texttt{Error} \rightarrow \texttt{OK}(S) \\ \quad \mid \texttt{OK}(C) \rightarrow \texttt{if } Validate(S, C) \texttt{ then OK}(C) \texttt{ else OK}(S) \end{array}$$

Theorem 1 *If Validate is a verified validator in the sense of definition 9, $Comp'$ and $Comp''$ are verified compilers in the sense of definition 8.*

Verification of a translation validator is therefore an attractive alternative to the verification of a compiler, provided the validator is smaller and simpler than the compiler.

In the presentation above, the validator receives unadorned source and compiled codes as arguments. In practice, the validator can also take advantage of additional information generated by the compiler and transmitted to the validator as part of C or separately. For instance, the validator of [87] exploits debugging information to suggest

[2] This conservatism doesn't necessarily render validators incomplete: a validator can be complete with respect to a particular code transformation or family of transformations.

a correspondence between program points and between variables of S and C. Credible compilation [86] carries this approach to the extreme: the compiler is supposed to annotate C with a full proof of $S \approx C$, so that translation validation reduces to proof checking.

2.2.3 Proof-carrying code and certifying compilers

The proof-carrying code (PCC) approach [75,2,33] does not attempt to establish semantic preservation between a source program and some compiled code. Instead, PCC focuses on the generation of independently-checkable evidence that the compiled code C satisfies a behavioral specification *Spec* such as type and memory safety. PCC makes use of a *certifying compiler*, which is a function *CComp* that either fails or returns both a compiled code C and a proof π of the property $C \models \mathit{Spec}$. The proof π, also called a *certificate*, can be checked independently by the code user; there is no need to trust the code producer, nor to formally verify the compiler itself.

In a naive view of PCC, the certificate π generated by the compiler is a full proof term and the client-side verifier is a general-purpose proof checker. In practice, it is sufficient to generate enough hints so that such a full proof can be reconstructed cheaply on the client side by a specialized checker [78]. If the property of interest is type safety, PCC can reduce to type-checking of compiled code, as in Java bytecode verification [90] or typed assembly language [72]: the certificate π reduces to type annotations, and the client-side verifier is a type checker.

In the original PCC design, the certifying compiler is specialized for a fixed property of programs (e.g. type and memory safety), and this property is simple enough to be established by the compiler itself. For richer properties, it becomes necessary to provide the certifying compiler with a certificate that the source program S satisfies the property. It is also possible to make the compiler generic with respect to a family of program properties. This extension of PCC is called proof-preserving compilation in [89] and certificate translation in [7,8].

In all cases, it suffices to formally verify the client-side checker to obtain guarantees as strong as those obtained from compiler verification. Symmetrically, a certifying compiler can be constructed (at least theoretically) from a verified compiler. Assume that *Comp* is a verified compiler, using definition 6 as our notion of semantic preservation, and further assume that the verification was conducted with a proof assistant that produces proof terms, such as Coq. Let Π be a proof term for the semantic preservation theorem of *Comp*, namely

$$\Pi : \forall S, C, \quad \mathit{Comp}(S) = \mathtt{OK}(C) \Longrightarrow S \models \mathit{Spec} \Longrightarrow C \models \mathit{Spec}$$

Via the Curry-Howard isomorphism, Π is a function that takes S, C, a proof of $\mathit{Comp}(S) = \mathtt{OK}(C)$ and a proof of $S \models \mathit{Spec}$, and returns a proof of $C \models \mathit{Spec}$. A certifying compiler of the proof-preserving kind can then be defined as follows:

$$\begin{array}{l} \mathit{CComp}(S : \mathit{Source}, \pi_s : S \models \mathit{Spec}) = \\ \quad \mathtt{match}\ \mathit{Comp}(S)\ \mathtt{with} \\ \quad \mid \mathtt{Error} \rightarrow \mathtt{Error} \\ \quad \mid \mathtt{OK}(C) \rightarrow \mathtt{OK}(C, \Pi\ S\ C\ \pi_{eq}\ \pi_s) \end{array}$$

(Here, π_{eq} is a proof term for the proposition $\mathit{Comp}(S) = \mathtt{OK}(C)$, which trivially holds in the context of the `match` above. Actually building this proof term in Coq requires

additional baggage in the definition above that we omitted for simplicity.) The accompanying client-side checker is the Coq proof checker. While the certificate produced by *CComp* is huge (it contains a proof of soundness for the compilation of all source programs, not just for S), it could perhaps be specialized for S and C using partial evaluation techniques.

2.3 Composition of compilation passes

Compilers are naturally decomposed into several passes that communicate through intermediate languages. It is fortunate that verified compilers can also be decomposed in this manner.

Let $Comp_1$ and $Comp_2$ be compilers from languages L_1 to L_2 and L_2 to L_3, respectively. Assume that the semantic preservation property $\approx$ is transitive. (This is true for all properties considered in section 2.1.) Consider the monadic composition of $Comp_1$ and $Comp_2$:

$$\begin{array}{l} Comp(S) = \\ \quad \texttt{match}\ Comp_1(S)\ \texttt{with} \\ \quad \mid \texttt{Error} \rightarrow \texttt{Error} \\ \quad \mid \texttt{OK}(I) \rightarrow Comp_2(I) \end{array}$$

Theorem 2 *If the compilers $Comp_1$ and $Comp_2$ are verified, so is their monadic composition $Comp$.*

2.4 Summary

The conclusions of this discussion are simple and define the methodology we have followed to verify the Compcert compiler back-end.

1. Provided the target language of the compiler has deterministic semantics, an appropriate specification for the soundness proof of the compiler is the combination of definitions 5 (forward simulation for safe source programs) and 8 (verified compiler), namely
$$\forall S, C, B \notin \texttt{Wrong}, \quad Comp(S) = \texttt{OK}(C) \ \wedge \ S \Downarrow B \Longrightarrow C \Downarrow B \qquad (i)$$
2. A verified compiler can be structured as a composition of compilation passes, as is commonly done for conventional compilers. Each pass can be proved sound independently. However, all intermediate languages must be given appropriate formal semantics.
3. For each pass, we have a choice between proving the code that implements this pass or performing the transformation via untrusted code, then verifying its results using a verified validator. The latter approach can reduce the amount of code that needs to be proved. In our experience, the verified validator approach is particularly effective for advanced optimizations, but less so for nonoptimizing translation passes and basic dataflow optimizations. Therefore, we did not use this approach for the compilation passes presented in this article, but elected to prove directly the soundness of these passes. [3]

[3] However, a posteriori validation with a verified validator is used for some auxiliary heuristics such as graph coloring during register allocation (section 8.2) and node enumeration during CFG linearization (section 10.2).

4. Finally, provided the proof of (i) is carried out in a prover such as Coq that generates proof terms and follows the Curry-Howard isomorphism, it is at least theoretically possible to use the verified compiler in a context of proof-carrying code.

3 Infrastructure

This section describes elements of syntax, semantics and proofs that are used throughout the Compcert development.

3.1 Programs

The syntax of programs in the source, intermediate and target languages share the following common shape.

Programs:

$$\begin{array}{lll} P & ::= \{\ \texttt{vars} = id_1 = data_1^*; \ldots id_n = data_n^*; & \text{global variables} \\ & \quad \texttt{functs} = id_1 = Fd_1; \ldots id_n = Fd_n; & \text{functions} \\ & \quad \texttt{main} = id\ \} & \text{entry point} \end{array}$$

Function definitions:

$$Fd \quad ::= \texttt{internal}(F) \mid \texttt{external}(Fe)$$

Definitions of internal functions:

$$F \quad ::= \{\ \texttt{sig} = sig; \texttt{body} = \ldots; \ldots\ \} \qquad \text{(language-dependent)}$$

Declarations of external functions:

$$Fe \quad ::= \{\ \texttt{tag} = id; \texttt{sig} = sig\ \}$$

Initialization data for global variables:

$$\begin{array}{ll} data & ::= \texttt{reserve}(n) \mid \texttt{int8}(n) \mid \texttt{int16}(n) \\ & \quad \mid \texttt{int32}(n) \mid \texttt{float32}(f) \mid \texttt{float64}(f) \end{array}$$

Function signatures:

$$sig \quad ::= \{\ \texttt{args} = \vec{\tau};\ \texttt{res} = (\tau \mid \texttt{void})\}$$

Types:

$$\begin{array}{lll} \tau & ::= \texttt{int} & \text{integers and pointers} \\ & \quad \mid \texttt{float} & \text{floating-point numbers} \end{array}$$

A program is composed of a list of global variables with their initialization data, a list of functions, and the name of a distinguished function that constitutes the program entry point (like `main` in C). Initialization data is a sequence of integer or floating-point constants in various sizes, or $\texttt{reserve}(n)$ to denote n bytes of uninitialized storage.

Two kinds of function definitions Fd are supported. Internal functions F are defined within the program. The precise contents of an internal function depends on the language considered, but include at least a signature sig giving the number and types of parameters and results and a body defining the computation (e.g. as a statement in Cminor or a list of instructions in PPC). An external function Fe is not defined within the program, but merely declared with an external name and a signature. External functions are intended to model input/output operations or other kinds of system calls. The observable behavior of the program will be defined in terms of a trace of invocations of external functions (see section 3.4).

The types τ used in function signatures and in other parts of Compcert are extremely coarse: we only distinguish between integers or pointers on the one hand (type `int`) and floating-point numbers on the other hand (type `float`). In particular, we make no attempt to track the type of data pointed to by a pointer. These "types" are best thought of as hardware register classes. Their main purpose is to guide register allocation and help determine calling conventions from the signature of the function being called.

Each compilation pass is presented as a total function $\texttt{transf} : F_1 \to (\texttt{OK}(F_2) \mid \texttt{Error}(msg))$ where F_1 and F_2 are the types of internal functions for the source and target languages (respectively) of the compilation pass. Such transformation functions are generically extended to function definitions by taking $\texttt{transf}(Fe) = \texttt{OK}(Fe)$, then to whole programs as a monadic "map" operation over function definitions:

$$\texttt{transf}(P) = \texttt{OK}\ \{\texttt{vars} = P.\texttt{vars}; \texttt{functs} = (\ldots\ id_i = Fd'_i; \ldots); \texttt{main} = P.\texttt{main}\}$$

if and only if $P.\texttt{functs} = (\ldots\ id_i = Fd_i; \ldots)$ and $\texttt{transf}(Fd_i) = \texttt{OK}(Fd'_i)$ for all i.

3.2 Values and memory states

The dynamic semantics of the Compcert languages manipulate values that are the discriminated union of 32-bit integers, 64-bit IEEE double precision floats, pointers, and a special `undef` value denoting in particular the contents of uninitialized memory. Pointers are composed of a block identifier b and a signed byte offset δ within this block.

Values:	$v ::= \texttt{int}(n)$	32-bit machine integer
	$\mid \texttt{float}(f)$	64-bit floating-point number
	$\mid \texttt{ptr}(b, \delta)$	pointer
	$\mid \texttt{undef}$	
Memory blocks:	$b \in \mathbb{Z}$	block identifiers
Block offsets:	$\delta ::= n$	byte offset within a block (signed)

Values are assigned types in the obvious manner:

$$\texttt{int}(n) : \texttt{int} \qquad \texttt{float}(f) : \texttt{float} \qquad \texttt{ptr}(b, \delta) : \texttt{int} \qquad \texttt{undef} : \tau \text{ for all } \tau$$

The memory model used in our semantics is detailed in [59]. Memory states M are modeled as collections of blocks separated by construction and identified by (mathematical) integers b. Each block has lower and upper bounds $\mathcal{L}(M, b)$ and $\mathcal{H}(M, b)$, fixed at allocation time, and associates values to byte offsets $\delta \in [\mathcal{L}(M, b), \mathcal{H}(M, b))$. The basic operations over memory states are:

- $\texttt{alloc}(M, l, h) = (b, M')$: allocate a fresh block with bounds $[l, h)$, of size $(h - l)$ bytes; return its identifier b and the updated memory state M'.
- $\texttt{store}(M, \kappa, b, \delta, v) = \lfloor M' \rfloor$: store value v in the memory quantity κ of block b at offset δ; return update memory state M'.
- $\texttt{load}(M, \kappa, b, \delta) = \lfloor v \rfloor$: read the value v contained in the memory quantity κ of block b at offset δ.
- $\texttt{free}(M, b) = M'$: free (invalidate) the block b and return the updated memory M'.

The memory quantities κ involved in `load` and `store` operations represent the kind, size and signedness of the datum being accessed:

Memory quantities: $\kappa ::= \texttt{int8signed} \mid \texttt{int8unsigned} \mid \texttt{int16signed} \mid \texttt{int16unsigned} \mid \texttt{int32} \mid \texttt{float32} \mid \texttt{float64}$

The `load` and `store` operations may fail when given an invalid block b or an out-of-bounds offset δ. Therefore, they return option types, with $\lfloor v \rfloor$ (read: "some v") denoting success with result v, and $\emptyset$ (read: "none") denoting failure. In this particular instance of the memory model of [59], `alloc` and `free` never fail. In particular, this means that we assume an infinite memory. This design decision is discussed further in section 17.4.

The four operations of the memory model satisfy a number of algebraic properties stated and proved in [59]. The following "load-after-store" property gives the general flavor of the memory model. Assume $\texttt{store}(M_1, \kappa, b, \delta, v) = \lfloor M_2 \rfloor$ and $\texttt{load}(M_1, \kappa', b', \delta') = \lfloor v' \rfloor$. Then,

$$\texttt{load}(M_2, \kappa', b', \delta') = \begin{cases} \lfloor \texttt{cast}(v, \kappa') \rfloor & \text{if } b' = b \text{ and } \delta' = \delta \text{ and } |\kappa'| = |\kappa|; \\ \lfloor v' \rfloor & \text{if } b' \neq b \text{ or } \delta + |\kappa| \leq \delta' \text{ or } \delta' + |\kappa'| \leq \delta; \\ \lfloor \texttt{undef} \rfloor & \text{otherwise.} \end{cases}$$

The $\texttt{cast}(v, \kappa')$ function performs truncation or sign-extension of value v as prescribed by the quantity κ'. Note that `undef` is returned (instead of a machine-dependent value) in cases where the quantities κ and κ' used for writing and reading disagree, or in cases where the ranges of bytes written $[\delta, \delta + |\kappa|)$ and read $[\delta', \delta' + |\kappa'|)$ partially overlap. This way, the memory model hides the endianness and bit-level representations of integers and floats and makes it impossible to forge pointers from sequences of bytes [59, section 7].

3.3 Global environments

The Compcert languages support function pointers but follow a "Harvard" model where functions and data reside in different memory spaces, and the memory space for functions is read-only (no self-modifying code). We use positive block identifiers b to refer to data blocks and negative b to refer to functions via pointers. The operational semantics for the Compcert languages are parameterized by a global environment G that does not change during execution. A global environment G maps function blocks $b < 0$ to function definitions. Moreover, it maps global identifiers (of functions or global variables) to blocks b. The basic operations over global environments are:

- $\texttt{funct}(G, b) = \lfloor Fd \rfloor$: return the function definition Fd corresponding to the block $b < 0$, if any.
- $\texttt{symbol}(G, id) = \lfloor b \rfloor$: return the block b corresponding to the global variable or function name id, if any.
- $\texttt{globalenv}(P) = G$: construct the global environment G associated with the program P.
- $\texttt{initmem}(P) = M$: construct the initial memory state M for executing the program P.

The $\texttt{globalenv}(P)$ and $\texttt{initmem}(P)$ functions model (at a high level of abstraction) the operation of a linker and a program loader. Unique, positive blocks b are allocated and associated to each global variable ($id = data^*$) of P, and the contents of these

blocks are initialized according to $data^*$. Likewise, unique, negative blocks b are associated to each function definition $(id = Fd)$ of P. In particular, if the functions of P have unique names, the following equivalence holds:

$$(id, Fd) \in P.\texttt{functs} \iff \exists b < 0.\ \texttt{symbol}(\texttt{globalenv}(P), id) = \lfloor b \rfloor \\ \wedge\ \texttt{funct}(\texttt{globalenv}(P), b) = \lfloor Fd \rfloor$$

The allocation of blocks for functions and global variables is deterministic so that convenient commutation properties hold between operations on global environments and per-function transformations of programs as defined in section 3.1.

Lemma 1 *Assume* $\texttt{transf}(P) = \texttt{OK}(P')$.

- $\texttt{initmem}(P') = \texttt{initmem}(P)$.
- *If* $\texttt{symbol}(\texttt{globalenv}(P), id) = \lfloor b \rfloor$, *then* $\texttt{symbol}(\texttt{globalenv}(P'), id) = \lfloor b \rfloor$.
- *If* $\texttt{funct}(\texttt{globalenv}(P), b) = \lfloor Fd \rfloor$, *then there exists a function definition* Fd' *such that* $\texttt{funct}(\texttt{globalenv}(P'), b) = \lfloor Fd' \rfloor$ *and* $\texttt{transf}(Fd) = \texttt{OK}(Fd')$.

3.4 Traces

We express the observable behaviors of programs in terms of traces of input-output events, each such event corresponding to an invocation of an external function. An event records the external name of the external function, the values of the arguments provided by the program, and the return value provided by the environment (e.g. the operating system).

Events:	$\nu ::= id(\vec{v_\nu} \mapsto v_\nu)$	
Event values:	$v_\nu ::= \texttt{int}(n) \mid \texttt{float}(f)$	
Traces:	$t ::= \epsilon \mid \nu.t$	finite traces (inductive)
	$T ::= \epsilon \mid \nu.T$	finite or infinite traces (coinductive)
Behaviors:	$B ::= \texttt{converges}(t, n)$	termination with trace t and exit code n
	$\mid \texttt{diverges}(T)$	divergence with trace T
	$\mid \texttt{goeswrong}(t)$	going wrong with trace t

We consider two types of traces: finite traces t for terminating or "going wrong" executions and finite or infinite traces T for diverging executions. Note that a diverging program can generate an empty or finite trace of input-output events (think infinite empty loop).

Concatenation of a finite trace t and a finite trace t' or infinite trace T is written $t.t'$ or $t.T$. It is associative and admits the empty trace ϵ as neutral element.

The values that are arguments and results of input-output events are required to be integers or floats. Since external functions cannot modify the memory state, passing them pointer values would be useless. Even with this restriction, events and traces can still model character-based input-output. We encapsulate these restrictions in the following inference rule that defines the effect of applying an external function Fe to arguments $\vec{v}$.

$$\frac{\begin{array}{c}\vec{v} \text{ and } v \text{ are integers or floats}\\ \vec{v} \text{ and } v \text{ agree in number and types with } Fe.\texttt{sig}\\ t = Fe.\texttt{tag}(\vec{v} \mapsto v)\end{array}}{\vdash Fe(\vec{v}) \stackrel{t}{\Rightarrow} v}$$

Note that the result value v and therefore the trace t are not completely determined by this rule. We return to this point in section 13.3.

3.5 Transition semantics

The operational semantics for the source, target and intermediate languages of the Compcert back-end are defined as labeled transition systems. The transition relation for each language is written $G \vdash S \stackrel{t}{\rightarrow} S'$ and denotes one execution step from state S to state S' in global environment G. The trace t denotes the observable events generated by this execution step. Transitions corresponding to an invocation of an external function record the associated event in t. Other transitions have $t = \epsilon$. In addition to the type of states S and the transition relation $G \vdash S \stackrel{t}{\rightarrow} S'$, each language defines two predicates:

- $\texttt{initial}(P, S)$: the state S is an initial state for the program P. Typically, S corresponds to an invocation of the main function of P in the initial memory state $\texttt{initmem}(P)$.
- $\texttt{final}(S, n)$: the state S is a final state with exit code n. Typically, this means that the program is returning from the initial invocation of its main function, with return value $\texttt{int}(n)$.

Executions are modeled classically as sequences of transitions from an initial state to a final state. We write $G \vdash S \stackrel{t}{\rightarrow}{}^{+} S'$ to denote one or several transitions (transitive closure), $G \vdash S \stackrel{t}{\rightarrow}{}^{*} S'$ to denote zero, one or several transitions (reflexive transitive closure), and $G \vdash S \stackrel{T}{\rightarrow} \infty$ to denote an infinite sequence of transitions starting with S. The traces t (finite) and T (finite or infinite) are formed by concatenating the traces of elementary transitions. Formally:

$$G \vdash S \stackrel{\epsilon}{\rightarrow}{}^{*} S \qquad \frac{G \vdash S \stackrel{t_1}{\rightarrow} S' \quad G \vdash S' \stackrel{t_2}{\rightarrow}{}^{*} S''}{G \vdash S \stackrel{t_1.t_2}{\rightarrow}{}^{*} S''}$$

$$\frac{G \vdash S \stackrel{t_1}{\rightarrow} S' \quad G \vdash S' \stackrel{t_2}{\rightarrow}{}^{*} S''}{G \vdash S \stackrel{t_1.t_2}{\rightarrow}{}^{+} S''} \qquad \frac{\frac{G \vdash S \stackrel{t}{\rightarrow} S' \quad G \vdash S' \stackrel{T}{\rightarrow} \infty}{}}{G \vdash S \stackrel{t.T}{\rightarrow} \infty}$$

As denoted by the double horizontal bar, the inference rule defining $G \vdash S \stackrel{T}{\rightarrow} \infty$ is to be interpreted coinductively, as a greatest fixpoint. The observable behavior of a program P is defined as follows. Starting from an initial state, if a finite sequence of reductions with trace t leads to a final state with exit code n, the program has observable behavior $\texttt{converges}(t, n)$. If an infinite sequence of reductions with trace T is possible, the observable behavior of the program is $\texttt{diverges}(T)$. Finally, if the program gets stuck on a non-final state after performing a sequence of reductions with trace t, the behavior is $\texttt{goeswrong}(t)$.

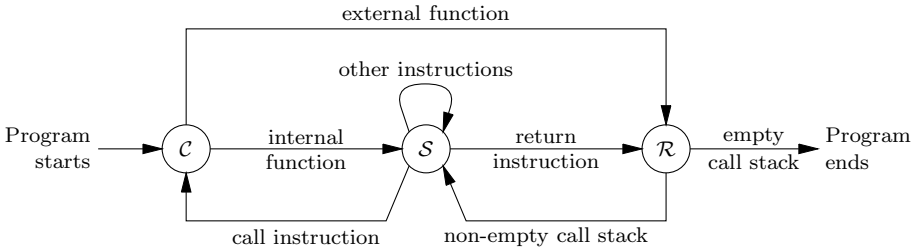

Fig. 3 Transitions between the three kinds of program states.

$$\frac{\texttt{initial}(P,S) \quad \texttt{globalenv}(P) \vdash S \stackrel{t}{\rightarrow}{}^{*} S' \quad \texttt{final}(S',n)}{P \Downarrow \texttt{converges}(t,n)}$$

$$\frac{\texttt{initial}(P,S) \quad \texttt{globalenv}(P) \vdash S \stackrel{T}{\rightarrow} \infty}{P \Downarrow \texttt{diverges}(T)}$$

$$\frac{\texttt{initial}(P,S) \quad \texttt{globalenv}(P) \vdash S \stackrel{t}{\rightarrow}{}^{*} S' \quad S' \not\rightarrow \quad \forall n,\ \neg\texttt{final}(S',n)}{P \Downarrow \texttt{goeswrong}(t)}$$

The set of "going wrong" behaviors is defined in the obvious manner: $\texttt{Wrong} = \{\texttt{goeswrong}(t) \mid t \text{ a finite trace}\}$.

3.6 Program states

The contents of a program state vary from language to language. For the assembly language PPC, a state is just a pair of a memory state and a mapping from processor registers to values (section 13.2). For the other languages of the Compcert back-end, states come in three kinds written $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{R}$.

- Regular states $\mathcal{S}$ correspond to an execution point within an internal function. They carry the function in question and a program point within this function, possibly along with additional language-specific components such as environments giving values to function-local variables.
- Call states $\mathcal{C}$ materialize parameter passing from the caller to the callee. They carry the function definition Fd being invoked and either a list of argument values or an environment where the argument values can be found at conventional locations.
- Return states $\mathcal{R}$ correspond to returning from a function to its caller. They carry at least the return value or an environment where this value can be found.

All three kinds of states also carry the current memory state as well as a call stack: a list of frames describing the functions in the call chain, with the corresponding program points where execution should be resumed on return, possibly along with function-local environments.

If we project the transition relation on the three-element set $\{\mathcal{S},\mathcal{C},\mathcal{R}\}$, abstracting away the components carried by the states, we obtain the finite automaton depicted in figure 3. This automaton is shared by all languages of the Compcert back-end except PPC, and it illustrates the interplay between the three kinds of states. Initial states

are call states with empty call stacks. A call state where the called function is external transitions directly to a return state after generating the appropriate event in the trace. A call state where the called function is internal transitions to a regular state corresponding to the function entry point, possibly after binding the argument values to the parameter variables. Non-call, non-return instructions go from regular states to regular states. A non-tail call instruction resolves the called function, pushes a return frame on the call stack and transitions to the corresponding call state. A tail call is similar but does not push a return frame. A return instruction transitions to a return state. A return state with a non-empty call stack pops the top return frame and moves to the corresponding regular state. A return state with an empty call stack is a final state.

3.7 Generic simulation diagrams

Consider two languages L_1 and L_2 defined by their transition semantics as described in section 3.5. Let P_1 be a program in L_1 and P_2 a program in L_2 obtained by applying a transformation to P_1. We wish to show that P_2 preserves the semantics of P_1, that is, $P_1 \Downarrow B \Longrightarrow P_2 \Downarrow B$ for all behaviors $B \notin \texttt{Wrong}$. The approach we use throughout this work is to construct a relation $S_1 \sim S_2$ between states of L_1 and states of L_2 and show that it is a forward simulation. First, initial states and final states should be related by $\sim$ in the following sense:

- Initial states: if $\texttt{initial}(P_1, S_1)$ and $\texttt{initial}(P_2, S_2)$, then $S_1 \sim S_2$.
- Final states: if $S_1 \sim S_2$ and $\texttt{final}(S_1, n)$, then $\texttt{final}(S_2, n)$.

Second, assuming $S_1 \sim S_2$, we need to relate transitions starting from S_1 in L_1 with transitions starting from S_2 in L_2. The simplest property that guarantees semantic preservation is the following lock-step simulation property:

Definition 10 Lock-step simulation: if $S_1 \sim S_2$ and $G_1 \vdash S_1 \xrightarrow{t} S_1'$, there exists S_2' such that $G_2 \vdash S_2 \xrightarrow{t} S_2'$ and $S_1' \sim S_2'$.

(G_1 and G_2 are the global environments corresponding to P_1 and P_2, respectively.) Figure 4, top left, shows the corresponding diagram.

Theorem 3 *Under hypotheses "initial states", "final states" and "lock-step simulation", $P_1 \Downarrow B$ and $B \notin \texttt{Wrong}$ imply $P_2 \Downarrow B$.*

Proof A trivial induction shows that $S_1 \sim S_2$ and $G_1 \vdash S_1 \xrightarrow{t}{}^* S_1'$ implies the existence of S_2' such that $G_2 \vdash S_2 \xrightarrow{t}{}^* S_2'$ and $S_1' \sim S_2'$. Likewise, a trivial coinduction shows that $S_1 \sim S_2$ and $G_1 \vdash S_1 \xrightarrow{T} \infty$ implies $G_2 \vdash S_2 \xrightarrow{T} \infty$. The result follows from the definition of $\Downarrow$.

The lock-step simulation hypothesis is too strong for many program transformations of interest, however. Some transformations cause transitions in P_1 to disappear in P_2, e.g. removal of no-operations, elimination of redundant computations, or branch tunneling. Likewise, some transformations introduce additional transitions in P_2, e.g. insertion of spilling and reloading code. Naively, we could try to relax the simulation hypothesis as follows:

Definition 11 Naive "star" simulation: if $S_1 \sim S_2$ and $G_1 \vdash S_1 \xrightarrow{t} S_1'$, there exists S_2' such that $G_2 \vdash S_2 \xrightarrow{t}{}^* S_2'$ and $S_1' \sim S_2'$.

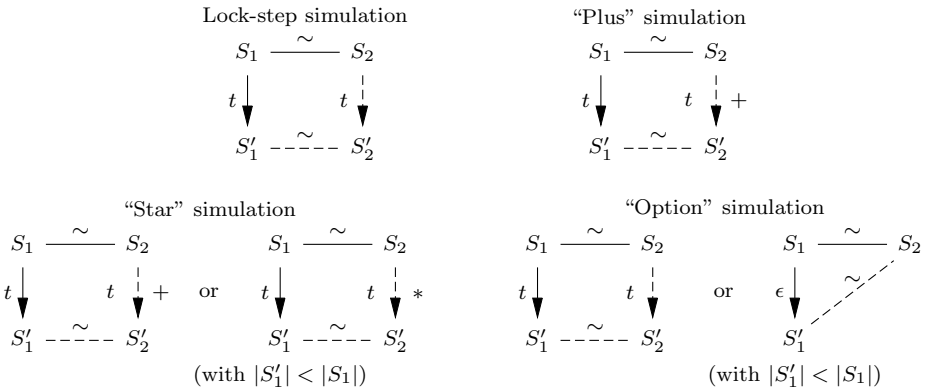

Fig. 4 Four kinds of simulation diagrams that imply semantic preservation. Solid lines denote hypotheses; dashed lines denote conclusions.

This hypothesis suffices to show the preservation of terminating behaviors, but does not guarantee that diverging behaviors are preserved because of the classic "infinite stuttering" problem. The original program P_1 could perform infinitely many silent transitions $S_1 \stackrel{\epsilon}{\to} S_2 \stackrel{\epsilon}{\to} \ldots \stackrel{\epsilon}{\to} S_n \stackrel{\epsilon}{\to} \ldots$ while the transformed program P_2 is stuck in a state S' such that $S_i \sim S'$ for all i. In this case, P_1 diverges while P_2 does not, and semantic preservation does not hold. To rule out the infinite stuttering problem, assume we are given a measure $|S_1|$ over the states of language L_1. This measure ranges over a type $\mathcal{M}$ equipped with a well-founded ordering $<$ (that is, there are no infinite decreasing chains of elements of $\mathcal{M}$). We require that the measure strictly decreases in cases where stuttering could occur, making it impossible for stuttering to occur infinitely.

Definition 12 "Star" simulation: if $S_1 \sim S_2$ and $G_1 \vdash S_1 \stackrel{t}{\to} S_1'$, either

1. there exists S_2' such that $G_2 \vdash S_2 \stackrel{t}{\to}{}^+ S_2'$ and $S_1' \sim S_2'$,
2. or $|S_1'| < |S_1|$ and there exists S_2' such that $G_2 \vdash S_2 \stackrel{t}{\to}{}^* S_2'$ and $S_1' \sim S_2'$.

Diagrammatically, this hypothesis corresponds to the bottom left part of figure 4. (Equivalently, part 2 of the definition could be replaced by "or $|S_1'| < |S_1|$ and $t = \epsilon$ and $S_2 \sim S_1'$", but the formulation above is more convenient in practice.)

Theorem 4 *Under hypotheses "initial states", "final states" and "star simulation", $P_1 \Downarrow B$ and $B \notin$ `Wrong` imply $P_2 \Downarrow B$.*

Proof A trivial induction shows that $S_1 \sim S_2$ and $G_1 \vdash S_1 \stackrel{t}{\to}{}^* S_1'$ implies the existence of S_2' such that $G_2 \vdash S_2 \stackrel{t}{\to}{}^* S_2'$ and $S_1' \sim S_2'$. This implies the desired result if B is a terminating behavior. For diverging behaviors, we first define (coinductively) the following "measured" variant of the $G_2 \vdash S_2 \stackrel{T}{\to} \infty$ relation:

$$\frac{G_2 \vdash S_2 \stackrel{t}{\to}{}^+ S_2' \qquad G_2 \vdash S_2', \mu' \stackrel{T}{\to} \infty}{G_2 \vdash S_2, \mu \stackrel{t.T}{\to} \infty}$$

$$\frac{G_2 \vdash S_2 \stackrel{t}{\to}{}^* S_2' \qquad \mu' < \mu \qquad G_2 \vdash S_2', \mu' \stackrel{T}{\to} \infty}{G_2 \vdash S_2, \mu \stackrel{t.T}{\to} \infty}$$

The second rule permits a number of potentially stuttering steps to be taken, provided the measure μ strictly decreases. After a finite number of invocations of this rule, it becomes non applicable and the first rule must be applied, forcing at least one transition to be taken and resetting the measure to an arbitrarily-chosen value. A straightforward coinduction shows that $G_1 \vdash S_1 \stackrel{T}{\rightarrow} \infty$ and $S_1 \sim S_2$ implies $G_2 \vdash S_2, |S_1| \stackrel{T}{\rightarrow} \infty$. To conclude, it suffices to prove that $G_2 \vdash S_2, \mu \stackrel{T}{\rightarrow} \infty$ implies $G_2 \vdash S_2 \stackrel{T}{\rightarrow} \infty$. This follows by coinduction and the following inversion lemma, proved by Noetherian induction over μ: if $G_2 \vdash S_2, \mu \stackrel{T}{\rightarrow} \infty$, there exists S_2', μ', t and T' such that $G_2 \vdash S_2 \stackrel{t}{\rightarrow} S_2'$ and $G_2 \vdash S_2', \mu' \stackrel{T'}{\rightarrow} \infty$ and $T = t.T'$.

Here are two stronger variants of the "star" simulation hypothesis that are convenient in practice. (See figure 4 for the corresponding diagrams.)

Definition 13 "Plus" simulation: if $S_1 \sim S_2$ and $G_1 \vdash S_1 \stackrel{t}{\rightarrow} S_1'$, there exists S_2' such that $G_2 \vdash S_2 \stackrel{t}{\rightarrow}{}^{+} S_2'$ and $S_1' \sim S_2'$.

Definition 14 "Option" simulation: if $S_1 \sim S_2$ and $G_1 \vdash S_1 \stackrel{t}{\rightarrow} S_1'$, either

1. there exists S_2' such that $G_2 \vdash S_2 \stackrel{t}{\rightarrow} S_2'$ and $S_1' \sim S_2'$,
2. or $|S_1'| < |S_1|$ and $t = \epsilon$ and $S_1' \sim S_2$.

Either simulation hypothesis implies the "star" simulation property and therefore semantic preservation per theorem 4.

4 The source language: Cminor

The input language of our back-end is called Cminor. It is a simple, low-level imperative language, comparable to a stripped-down, typeless variant of C. Another source of inspiration was the C-- intermediate language of Peyton Jones et al. [81]. In the CompCert compilation chain, Cminor is the lowest-level language that is still processor independent; it is therefore an appropriate language to start the back-end part of the compiler.

4.1 Syntax

Cminor is, classically, structured in expressions, statements, functions and whole programs.

Expressions:

a	$::= id$	reading a local variable
	$\mid cst$	constant
	$\mid op_1(a_1)$	unary arithmetic operation
	$\mid op_2(a_1, a_2)$	binary arithmetic operation
	$\mid \kappa[a_1]$	memory read at address a_1
	$\mid a_1 \ ? \ a_2 : a_3$	conditional expression

Constants:

cst	$::= n \mid f$	integer or float literal
	$\mid$ `addrsymbol`(id)	address of a global symbol
	$\mid$ `addrstack`(δ)	address within stack data

Unary operators:

op_1 ::=	`negint` | `notint` | `notbool`	integer arithmetic
	| `negf` | `absf`	float arithmetic
	| `cast8u` | `cast8s` | `cast16u` | `cast16s`	zero and sign extensions
	| `singleoffloat`	float truncation
	| `intoffloat` | `intuoffloat`	float-to-int conversions
	| `floatofint` | `floatofintu`	int-to-float conversions

Binary operators:

op_2 ::=	`add` | `sub` | `mul` | `div` | `divu` | `mod` | `modu`	integer arithmetic
	| `and` | `or` | `xor` | `shl` | `shr` | `shru`	integer bit operation
	| `addf` | `subf` | `mulf` | `divf`	float arithmetic
	| `cmp`(c) | `cmpu`(c) | `cmpf`(c)	comparisons

Comparisons:

c ::= `eq` \| `ne` \| `gt` \| `ge` \| `lt` \| `le`

Expressions are pure: side-effecting operations such as assignment and function calls are statements, not expressions. All arithmetic and logical operators of C are supported. Unlike in C, there is no overloading nor implicit conversions between types: distinct arithmetic operators are provided over integers and over floats; likewise, explicit conversion operators are provided to convert between floats and integers and perform zero and sign extensions. Memory loads are explicit and annotated with the memory quantity κ being accessed.

Statements:

s	::= `skip`	no operation
	| $id = a$	assignment to a local variable
	| $\kappa[a_1] = a_2$	memory write at address a_1
	| $id^? = a(\vec{a}) : sig$	function call
	| `tailcall` $a(\vec{a}) : sig$	function tail call
	| `return`$(a^?)$	function return
	| $s_1; s_2$	sequence
	| `if`$(a)\,\{s_1\}$ `else` $\{s_2\}$	conditional
	| `loop`$\,\{s_1\}$	infinite loop
	| `block`$\,\{s_1\}$	block delimiting `exit` constructs
	| `exit`(n)	terminate the $(n+1)^{\text{th}}$ enclosing block
	| `switch`$(a)\,\{tbl\}$	multi-way test and exit
	| $lbl : s$	labeled statement
	| `goto` lbl	jump to a label

Switch tables:

tbl ::=	`default` : `exit`(n)
	| `case` i : `exit`(n); tbl

Base statements are `skip`, assignment $id = a$ to a local variable, memory store $\kappa[a_1] = a_2$ (of the value of a_2 in the quantity κ at address a_1), function call (with optional assignment of the return value to a local variable), function tail call, and function return (with an optional result). Function calls are annotated with the signatures sig expected for the called function. A tail call `tailcall` $a(\vec{a})$ is almost equivalent to a regular call immediately followed by a `return`, except that the tail call deallocates the

```
double average(int arr[], int sz)
{
  double s; int i;
  for (i = 0, s = 0; i < sz; i++)
    s += arr[i];
  return s / sz;
}
```

```
"average"(arr, sz) : int, int -> float
{
  vars s, i;  stacksize 0;
  s = 0.0; i = 0;
  block { loop {
    if (i >= sz) exit(0);
    s = s +f floatofint(int32[arr + i*4]);
    i = i + 1;
  } }
  return s /f floatofint(sz);
}
```

Fig. 5 An example of a Cminor function (right) and the corresponding C code (left).

current stack data block before invoking the function. This enables tail recursion to execute in constant stack space.

Besides the familiar sequence $s_1; s_2$ and `if/then/else` constructs, control flow can be expressed either in an unstructured way using `goto` and labeled statements or in a structured way using infinite loops and the `block/exit` construct. `exit`(n) where $n \geq 0$ branches to the end of the $(n+1)^{\text{th}}$ enclosing `block` construct. The `switch`$(a)\,\{tbl\}$ construct matches the integer value of a against the cases listed in *tbl* and performs the corresponding `exit`. Appropriate nesting of a `switch` within several `block` constructs suffices to express C-like structured `switch` statements with fall-through behavior.

Internal functions: $F ::= \{$

`sig` $= sig;$	function signature
`params` $= \vec{id};$	parameters
`vars` $= \vec{id};$	local variables
`stacksize` $= n;$	size of stack data in bytes
`body` $= s\ \}$	function body

In addition to a parameter list, local variable declarations and a function body (a statement), a function definition comprises a type signature *sig* and a declaration of how many bytes of stack-allocated data it needs. A Cminor local variable does not reside in memory, and its address cannot be taken. However, the Cminor producer can explicitly stack-allocate some data (such as, in C, arrays and scalar variables whose addresses are taken). A fresh memory block of size F.`stacksize` is allocated each time F is invoked and automatically freed when it returns. The `addrstack`(δ) nullary operator returns a pointer within this block at byte offset δ.

Figure 5 shows a simple C function and the corresponding Cminor function, using an ad-hoc concrete syntax for Cminor. Both functions compute the average value of an array of integers, using float arithmetic. Note the explicit address computation `int32[tbl + i*4]` to access element `i` of the array, as well as the explicit `floatofint` conversions. The `for` loop is expressed as an infinite loop wrapped in a `block`, so that `exit`(0) in the Cminor code behaves like the `break` statement in C.

4.2 Dynamic semantics

The dynamic semantics of Cminor is defined using a combination of natural semantics for the evaluation of expressions and a labeled transition system in the style of section 3.5 for the execution of statements and functions.

$$\frac{E(id) = \lfloor v \rfloor}{G, \sigma, E, M \vdash id \Rightarrow v} \qquad \frac{\texttt{eval\_constant}(G, \sigma, cst) = \lfloor v \rfloor}{G, \sigma, E, M \vdash cst \Rightarrow v}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow v_1 \quad \texttt{eval\_unop}(op_1, v_1) = \lfloor v \rfloor}{G, \sigma, E, M \vdash op_1(a_1) \Rightarrow v}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow v_1 \quad G, \sigma, E, M \vdash a_2 \Rightarrow v_2 \quad \texttt{eval\_binop}(op_2, v_1, v_2) = \lfloor v \rfloor}{G, \sigma, E, M \vdash op_2(a_1, a_2) \Rightarrow v}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow \texttt{ptr}(b, \delta) \quad \texttt{load}(M, \kappa, b, \delta) = \lfloor v \rfloor}{G, \sigma, E, M \vdash \kappa[a_1] \Rightarrow v}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow v_1 \quad \texttt{istrue}(v_1) \quad G, \sigma, E, M \vdash a_2 \Rightarrow v_2}{G, \sigma, E, M \vdash (a_1 \ ?\ a_2 : a_3) \Rightarrow v_2}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow v_1 \quad \texttt{isfalse}(v_1) \quad G, \sigma, E, M \vdash a_3 \Rightarrow v_3}{G, \sigma, E, M \vdash (a_1 \ ?\ a_2 : a_3) \Rightarrow v_3}$$

$$G, \sigma, E, M \vdash \epsilon : \epsilon \qquad \frac{G, \sigma, E, M \vdash a \Rightarrow v \quad G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v}}{G, \sigma, E, M \vdash a.\vec{a} \Rightarrow v.\vec{v}}$$

Evaluation of constants:

$$\begin{aligned}
\texttt{eval_constant}(G, \sigma, i) &= \lfloor \texttt{int}(i) \rfloor \\
\texttt{eval_constant}(G, \sigma, f) &= \lfloor \texttt{float}(f) \rfloor \\
\texttt{eval_constant}(G, \sigma, \texttt{addrsymbol}(id)) &= \texttt{symbol}(G, id) \\
\texttt{eval_constant}(G, \sigma, \texttt{addrstack}(\delta)) &= \lfloor \texttt{ptr}(\sigma, \delta) \rfloor
\end{aligned}$$

Evaluation of unary operators (selected cases):

$$\begin{aligned}
\texttt{eval_unop}(\texttt{negf}, \texttt{float}(f)) &= \lfloor \texttt{float}(-f) \rfloor \\
\texttt{eval_unop}(\texttt{notbool}, v) &= \lfloor \texttt{int}(0) \rfloor \text{ if } \texttt{istrue}(v) \\
\texttt{eval_unop}(\texttt{notbool}, v) &= \lfloor \texttt{int}(1) \rfloor \text{ if } \texttt{isfalse}(v)
\end{aligned}$$

Evaluation of binary operators (selected cases):

$$\begin{aligned}
\texttt{eval_binop}(\texttt{add}, \texttt{int}(n_1), \texttt{int}(n_2)) &= \lfloor \texttt{int}(n_1 + n_2) \rfloor \pmod{2^{32}} \\
\texttt{eval_binop}(\texttt{add}, \texttt{ptr}(b, \delta), \texttt{int}(n)) &= \lfloor \texttt{ptr}(b, \delta + n) \rfloor \pmod{2^{32}} \\
\texttt{eval_binop}(\texttt{addf}, \texttt{float}(f_1), \texttt{float}(f_2)) &= \lfloor \texttt{float}(f_1 + f_2) \rfloor
\end{aligned}$$

Truth values:

$$\begin{aligned}
\texttt{istrue}(v) &\stackrel{\text{def}}{=} v \text{ is } \texttt{ptr}(b, \delta) \text{ or } \texttt{int}(n) \text{ with } n \neq 0 \\
\texttt{isfalse}(v) &\stackrel{\text{def}}{=} v \text{ is } \texttt{int}(0)
\end{aligned}$$

Fig. 6 Natural semantics for Cminor expressions.

4.2.1 Evaluation of expressions

Figure 6 defines the big-step evaluation of expressions as the judgment $G, \sigma, E, M \vdash a \Rightarrow v$, where a is the expression to evaluate, v its value, σ the stack data block, E an environment mapping local variables to values, and M the current memory state. The evaluation rules are straightforward. Most of the semantics is in the definition of the auxiliary functions `eval_constant`, `eval_unop` and `eval_binop`, for which some

representative cases are shown. These functions can return $\emptyset$, causing the expression to be undefined, if for instance an argument of an operator is `undef` or of the wrong type. Some operators (`add`, `sub` and `cmp`) operate both on integers and on pointers.

4.2.2 Execution of statements and functions

The labeled transition system that defines the small-step semantics for statements and function invocations follows the general pattern shown in sections 3.5 and 3.6. Program states have the following shape:

Program states:	$S ::= \mathcal{S}(F, s, k, \sigma, E, M)$	regular state
	$\mid \mathcal{C}(Fd, \vec{v}, k, M)$	call state
	$\mid \mathcal{R}(v, k, M)$	return state
Continuations:	$k ::= \texttt{stop}$	initial continuation
	$\mid s; k$	continue with s, then do as k
	$\mid \texttt{endblock}(k)$	leave a `block`, then do as k
	$\mid \texttt{returnto}(id^?, F, \sigma, E, k)$	return to caller

Regular states $\mathcal{S}$ carry the currently-executing function F, the statement under consideration s, the block identifier for the current stack data σ, and the values E of local variables.

Following a proposal by Appel and Blazy [3], we use continuation terms k to encode both the call stack and the program point within F where the statement s under consideration resides. A continuation k records what needs to be done once s reduces to `skip`, `exit` or `return`. The `returnto` parts of k represent the call stack: they record the local states of the calling functions. The top part of k up to the first `returnto` corresponds to an execution context for s, represented inside-out in the style of a zipper [45]. For example, the continuation $s; \texttt{endblock}(\ldots)$ corresponds to the context $\texttt{block}\,\{\,[\,];\, s\,\}$.

Figures 7 and 8 list the rules defining the transition relation $G \vdash S \xrightarrow{t} S'$. The rules in figure 7 address transitions within the currently-executing function. They are roughly of three kinds:

- Execution of an atomic computation step. For example, the rule for assignments transitions from $id = a$ to `skip`.
- Focusing on the active part of the current statement. For example, the rule for sequences transitions from $(s_1; s_2)$ with continuation k to s_1 with continuation $s_2; k$.
- Resuming a continuation that was set apart in the continuation. For instance, one of the rules for `skip` transitions from `skip` with continuation $s; k$ to s with continuation k.

Two auxiliary functions over continuations are defined: $\texttt{callcont}(k)$ discards the local context part of the continuation k, and $\texttt{findlabel}(lbl, s, k)$ returns a pair (s', k') of the leftmost sub-statement of s labeled lbl and of a continuation k' that extends k with the context surrounding s'. The combination of these two functions in the rule for `goto` suffices to implement the branching behavior of `goto` statements.

Figure 8 lists the transitions involving call states and return states, and defines initial states and final states. The definitions follow the general pattern depicted in figure 3. In particular, initial states are call states to the "main" function of the program, with no arguments and the `stop` continuation; symmetrically, final states are return

$$G \vdash \mathcal{S}(F, \texttt{skip}, (s;k), \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s, k, \sigma, E, M)$$
$$G \vdash \mathcal{S}(F, \texttt{skip}, \texttt{endblock}(k), \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{skip}, k, \sigma, E, M)$$

$$\frac{G, \sigma, E, M \vdash a \Rightarrow v}{G \vdash \mathcal{S}(F, (id = a), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{skip}, k, \sigma, E\{id \leftarrow v\}, M)}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow \texttt{ptr}(b, \delta) \qquad G, \sigma, E, M \vdash a_2 \Rightarrow v \qquad \texttt{store}(M, \kappa, b, \delta, v) = \lfloor M' \rfloor}{G \vdash \mathcal{S}(F, (\kappa[a_1] = [a_2]), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{skip}, k, \sigma, E, M')}$$

$$G \vdash \mathcal{S}(F, (s_1; s_2), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s_1, (s_2; k), \sigma, E, M)$$

$$\frac{G, \sigma, E, M \vdash a \Rightarrow v \qquad \texttt{istrue}(v)}{G \vdash \mathcal{S}(F, (\texttt{if}(a)\{s_1\}\ \texttt{else}\ \{s_2\}), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s_1, k, \sigma, E, M)}$$

$$\frac{G, \sigma, E, M \vdash a \Rightarrow v \qquad \texttt{isfalse}(v)}{G \vdash \mathcal{S}(F, (\texttt{if}(a)\{s_1\}\ \texttt{else}\ \{s_2\}), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s_2, k, \sigma, E, M)}$$

$$G \vdash \mathcal{S}(F, \texttt{loop}\{s\}, k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s, (\texttt{loop}\{s\}; k), \sigma, E, M)$$
$$G \vdash \mathcal{S}(F, \texttt{block}\{s\}, k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s, \texttt{endblock}(k), \sigma, E, M)$$
$$G \vdash \mathcal{S}(F, \texttt{exit}(n), (s;k), \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{exit}(n), k, \sigma, E, M)$$
$$G \vdash \mathcal{S}(F, \texttt{exit}(0), \texttt{endblock}(k), \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{skip}, k, \sigma, E, M)$$
$$G \vdash \mathcal{S}(F, \texttt{exit}(n+1), \texttt{endblock}(k), \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{exit}(n), k, \sigma, E, M)$$

$$\frac{G, \sigma, E, M \vdash a \Rightarrow \texttt{int}(n)}{G \vdash \mathcal{S}(F, (\texttt{switch}(a)\{tbl\}), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{exit}(tbl(n)), k, \sigma, E, M)}$$

$$G \vdash \mathcal{S}(F, (lbl : s), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s, k, \sigma, E, M)$$

$$\frac{\texttt{findlabel}(lbl, F.\texttt{body}, \texttt{callcont}(k)) = \lfloor s', k' \rfloor}{G \vdash \mathcal{S}(F, \texttt{goto}\ lbl, k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s', k', \sigma, E, M)}$$

$$\texttt{callcont}(s;k) = \texttt{callcont}(k) \qquad \texttt{callcont}(\texttt{endblock}(k)) = \texttt{callcont}(k)$$
$$\texttt{callcont}(k) = k \text{ otherwise}$$

$$\texttt{findlabel}(lbl, (s_1; s_2), k) = \begin{cases} \texttt{findlabel}(lbl, s_1, (s_2; k)) & \text{if not } \emptyset; \\ \texttt{findlabel}(lbl, s_2, k) & \text{otherwise} \end{cases}$$
$$\texttt{findlabel}(lbl, \texttt{if}(a)\{s_1\}\ \texttt{else}\ \{s_2\}, k) = \begin{cases} \texttt{findlabel}(lbl, s_1, k) & \text{if not } \emptyset; \\ \texttt{findlabel}(lbl, s_2, k) & \text{otherwise} \end{cases}$$
$$\texttt{findlabel}(lbl, \texttt{loop}\{s\}, k) = \texttt{findlabel}(lbl, s, (\texttt{loop}\{s\}; k))$$
$$\texttt{findlabel}(lbl, \texttt{block}\{s\}, k) = \texttt{findlabel}(lbl, s, \texttt{endblock}(k))$$
$$\texttt{findlabel}(lbl, (lbl : s), k) = \lfloor s, k \rfloor$$
$$\texttt{findlabel}(lbl, (lbl' : s), k) = \texttt{findlabel}(lbl, s, k) \text{ if } lbl' \neq lbl$$

Fig. 7 Transition semantics for Cminor, part 1: statements.

states with the `stop` continuation. The rules for function calls require that the signature of the called function matches exactly the signature annotating the call; otherwise execution gets stuck. A similar requirement exists in the C standard and is essential to support signature-dependent calling conventions later in the compiler. Taking again a leaf from the C standard, functions are allowed to terminate by `return` without an argument or by falling through their bodies only if their return signatures are `void`.

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow \mathtt{ptr}(b, 0) \quad G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \mathtt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\mathtt{sig} = sig}{G \vdash \mathcal{S}(F, (id^? = a_1(\vec{a}) : sig), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{C}(Fd, \vec{v}, \mathtt{returnto}(id^?, F, \sigma, E, k), M)}$$

$$\frac{G, \sigma, E, M \vdash a_1 \Rightarrow \mathtt{ptr}(b, 0) \quad G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \mathtt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\mathtt{sig} = sig}{G \vdash \mathcal{S}(F, (\mathtt{tailcall}\ a_1(\vec{a}) : sig), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{C}(Fd, \vec{v}, \mathtt{callcont}(k), M)}$$

$$\frac{F.\mathtt{sig.res} = \mathtt{void} \quad k = \mathtt{returnto}(\ldots) \text{ or } k = \mathtt{stop}}{G \vdash \mathcal{S}(F, \mathtt{skip}, k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{R}(\mathtt{undef}, k, \mathtt{free}(M, \sigma))}$$

$$\frac{F.\mathtt{sig.res} = \mathtt{void}}{G \vdash \mathcal{S}(F, \mathtt{return}, k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{R}(\mathtt{undef}, \mathtt{callcont}(k), \mathtt{free}(M, \sigma))}$$

$$\frac{F.\mathtt{sig.res} \neq \mathtt{void} \quad G, \sigma, E, M \vdash a \Rightarrow v}{G \vdash \mathcal{S}(F, \mathtt{return}(a), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{R}(v, \mathtt{callcont}(k), \mathtt{free}(M, \sigma))}$$

$$\frac{\mathtt{alloc}(M, 0, F.\mathtt{stackspace}) = (\sigma, M') \quad E = [F.\mathtt{params} \leftarrow \vec{v}; F.\mathtt{vars} \leftarrow \mathtt{undef}]}{G \vdash \mathcal{C}(\mathtt{internal}(F), \vec{v}, k, M) \xrightarrow{\epsilon} \mathcal{S}(F, F.\mathtt{body}, k, \sigma, E, M')}$$

$$\frac{\vdash Fe(\vec{v}) \stackrel{t}{\Rightarrow} v \text{ (see section 3.4)}}{G \vdash \mathcal{C}(\mathtt{external}(Fe), \vec{v}, k, M) \xrightarrow{t} \mathcal{R}(v, k, M)}$$

$$G \vdash \mathcal{R}(v, \mathtt{returnto}(id^?, F, \sigma, E, k), M) \xrightarrow{\epsilon} \mathcal{S}(F, \mathtt{skip}, k, \sigma, E\{id^? \leftarrow v\}, M)$$

$$\frac{\mathtt{symbol}(\mathtt{globalenv}(P), P.\mathtt{main}) = \lfloor b \rfloor \quad \mathtt{funct}(\mathtt{globalenv}(P), b) = \lfloor Fd \rfloor}{\mathtt{initial}(P, \mathcal{C}(Fd, \epsilon, \mathtt{stop}, \mathtt{initmem}(P)))}$$

$$\mathtt{final}(\mathcal{R}(\mathtt{int}(n), \mathtt{stop}, M), n)$$

Fig. 8 Transition semantics for Cminor, part 2: functions, initial states, final states.

4.2.3 Alternate natural semantics for statements and functions

For some applications, it is convenient to have an alternate natural (big-step) operational semantics for Cminor. We have developed such a semantics for the fragment of Cminor that excludes `goto` and labeled statements. The big-step judgments for terminating executions have the following form:

$$\begin{aligned} G, \sigma \vdash s, E, M &\stackrel{t}{\Rightarrow} out, E', M' \quad \text{(statements)} \\ G \vdash Fd(\vec{v}), M &\stackrel{t}{\Rightarrow} v, M' \qquad \text{(function calls)} \end{aligned}$$

E' and M' are the local environment and the memory state at the end of the execution; t is the trace of events generated during execution. Following Huisman and Jacobs [46], the outcome *out* indicates how the statement s terminated: either normally by running to completion ($out = \mathtt{Normal}$); or prematurely by executing an `exit` statement ($out = \mathtt{Exit}(n)$), `return` statement ($out = \mathtt{Return}(v^?)$ where $v^?$ is the value of the optional argument to `return`), or `tailcall` statement ($out = \mathtt{Tailreturn}(v)$). Additionally, we followed the coinductive approach to natural semantics of Leroy and Grall [60] to define (coinductively) big-step judgments for diverging executions, of the form

$$\begin{aligned} G, \sigma \vdash s, E, M &\stackrel{T}{\Rightarrow} \infty \quad \text{(diverging statements)} \\ G \vdash Fd(\vec{v}), M &\stackrel{T}{\Rightarrow} \infty \text{ (diverging function calls)} \end{aligned}$$

The definitions of these judgments can be found in the Coq development.

Theorem 5 *The natural semantics of* Cminor *is correct with respect to its transition semantics:*

1. *If* $G \vdash Fd(\vec{v}), M \stackrel{t}{\Rightarrow} v, M'$*, then* $G \vdash \mathcal{C}(Fd, \vec{v}, k, M) \stackrel{t}{\rightarrow}{}^{*} \mathcal{R}(v, k, M')$ *for all continuations* k *such that* $k = \texttt{callcont}(k)$.
2. *If* $G \vdash Fd(\vec{v}), M \stackrel{T}{\Rightarrow} \infty$*, then* $G \vdash \mathcal{C}(Fd, \vec{v}, k, M) \stackrel{T}{\rightarrow} \infty$ *for all continuations* k.

4.3 Static typing

Cminor is equipped with a trivial type system having only two types: `int` and `float`. (Pointers have static type `int`.) Function definitions and function calls are annotated with signatures *sig* giving the number and types of arguments, along with optional result types. All operators are monomorphic; therefore, the types of local variables can be inferred from their uses and are not declared.

The primary purpose of this trivial type system is to facilitate later transformations (see sections 8 and 12); for this purpose, all intermediate languages of Compcert are equipped with similar `int`-or-`float` type systems. By themselves, these type systems are too weak to give type soundness properties (absence of run-time type errors). For example, performing an integer addition of two pointers or two `undef` values is statically well-typed but causes the program to get stuck. Likewise, calling a function whose signature differs from that given at the `call` site is a run-time error, undetected by the type system; its semantics are not defined and the compiler can (and does) generate incorrect code for this call. It is the responsibility of the Cminor producer to avoid these situations, e.g. by using a richer type system. Nevertheless, the Cminor type system enjoys a type preservation property: values of static type `int` are always integers, pointers or `undef`, and values of static type `float` are always floating-point numbers or `undef`. This weak soundness property plays a role in the correctness proofs of section 12.3.

5 Instruction selection

The first compilation pass of Compcert rewrites expressions to exploit the combined arithmetic operations and addressing modes of the target processor. To take better advantage of the processor's capabilities, reassociation of integer additions and multiplications is also performed, as well as a small amount of constant propagation.

5.1 The target language: CminorSel

The target language for this pass is CminorSel, a variant of Cminor that uses a different, processor-specific set of operators. Additionally, a syntactic class of condition expressions *ce* (expressions used only for their truth values) is introduced.

Expressions:

a	::=	id	reading a local variable
		$\mid op(\vec{a})$	operator application
		$\mid \texttt{load}(\kappa, mode, \vec{a})$	memory read
		$\mid ce\ ?\ a_1 : a_2$	conditional expression

Condition expressions:

ce	::=	$\texttt{true} \mid \texttt{false}$	
		$\mid cond(\vec{a})$	elementary test
		$\mid ce_1\ ?\ ce_2 : ce_3$	conditional condition

Operators (machine-specific):

op	::=	$n \mid f \mid \texttt{move} \mid \ldots$	most of Cminor operators
		$\mid \texttt{addi}_n \mid \texttt{rolm}_{n,m} \mid \ldots$	PPC combined operators

Addressing modes (machine-specific):

$mode$	::=	$\texttt{indexed}(n)$	indexed, immediate displacement
		$\mid \texttt{indexed2}$	indexed, register displacement
		$\mid \texttt{global}(id, \delta)$	address is $id + \delta$
		$\mid \texttt{based}(id, \delta)$	indexed, displacement is $id + \delta$
		$\mid \texttt{stack}(\delta)$	address is stack pointer $+ \delta$

Conditions (machine-specific):

$cond$	::=	$\texttt{comp}(c) \mid \texttt{compimm}(c, n)$	signed integer / pointer comparison
		$\mid \texttt{compu}(c) \mid \texttt{compuimm}(c, n)$	unsigned integer comparison
		$\mid \texttt{compf}(c)$	float comparison

Statements:

s	::=	$\texttt{store}(\kappa, mode, \vec{a}, a)$	memory write
		$\mid \texttt{if}(ce)\ \{s_1\}\ \texttt{else}\ \{s_2\}$	conditional statement
		$\mid \ldots$	as in Cminor

For the PowerPC, the machine-specific operators op include all Cminor nullary, unary and binary operators except `notint`, `mod` and `modu` (these need to be synthesized from other operators) and adds immediate forms of many integer operators, as well as a number of combined operators such as not-or, not-and, and rotate-and-mask. ($\texttt{rolm}_{n,m}$ is a left rotation by n bits followed by a logical "and" with m.) A memory load or store now carries an addressing mode $mode$ and a list of expressions $\vec{a}$, from which the address being addressed is computed. Finally, conditional expressions and conditional statements now take condition expressions ce as arguments instead of normal expressions a.

The dynamic semantics of CminorSel resembles that of Cminor, with the addition of a new evaluation judgment for condition expressions $G, \sigma, E, M \vdash ce \Rightarrow (\texttt{false} \mid \texttt{true})$. Figure 9 shows the main differences with respect to the Cminor semantics.

5.2 The code transformation

Instruction selection is performed by a bottom-up rewriting of expressions. For each Cminor operator op, we define a "smart constructor" function written $\overline{op}$ that takes CminorSel expressions as arguments, performs shallow pattern-matching over them to recognize combined or immediate operations, and returns the corresponding CminorSel

Evaluation of expressions:

$$\frac{G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \texttt{eval\_op}(G, \sigma, op, \vec{v}) = \lfloor v \rfloor}{G, \sigma, E, M \vdash op(\vec{a}) \Rightarrow v}$$

$$\frac{G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \texttt{eval\_mode}(G, \sigma, mode, \vec{v}) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{load}(M, \kappa, b, \delta) = \lfloor v \rfloor}{G, \sigma, E, M \vdash \texttt{load}(\kappa, mode, \vec{a}) \Rightarrow v}$$

$$\frac{G, \sigma, E, M \vdash c \Rightarrow \texttt{true} \quad G, \sigma, E, M \vdash a_1 \Rightarrow v_1}{G, \sigma, E, M \vdash (c \ ? \ a_1 : a_2) \Rightarrow v_1}$$

$$\frac{G, \sigma, E, M \vdash c \Rightarrow \texttt{false} \quad G, \sigma, E, M \vdash a_2 \Rightarrow v_2}{G, \sigma, E, M \vdash (c \ ? \ a_1 : a_2) \Rightarrow v_2}$$

Evaluation of condition expressions:

$$G, \sigma, E, M \vdash \texttt{true} \Rightarrow \texttt{true} \qquad G, \sigma, E, M \vdash \texttt{false} \Rightarrow \texttt{false}$$

$$\frac{G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \texttt{eval\_cond}(cond, \vec{v}) = \lfloor b \rfloor}{G, \sigma, E, M \vdash cond(\vec{a}) \Rightarrow b}$$

$$\frac{G, \sigma, E, M \vdash ce_1 \Rightarrow \texttt{true} \quad G, \sigma, E, M \vdash ce_2 \Rightarrow b}{G, \sigma, E, M \vdash (ce_1 \ ? \ ce_2 : ce_3) \Rightarrow b}$$

$$\frac{G, \sigma, E, M \vdash ce_1 \Rightarrow \texttt{false} \quad G, \sigma, E, M \vdash ce_3 \Rightarrow b}{G, \sigma, E, M \vdash (ce_1 \ ? \ ce_2 : ce_3) \Rightarrow b}$$

Execution of statements:

$$\frac{\begin{array}{c} G, \sigma, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \texttt{eval\_mode}(G, \sigma, mode, \vec{v}) = \lfloor \texttt{ptr}(b, \delta) \rfloor \\ G, \sigma, E, M \vdash a \Rightarrow v \quad \texttt{store}(M, \kappa, b, \delta, v) = \lfloor M' \rfloor \end{array}}{G \vdash \mathcal{S}(F, \texttt{store}(\kappa, mode, \vec{a}, a), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, \texttt{skip}, k, \sigma, E, M')}$$

$$\frac{G, \sigma, E, M \vdash ce \Rightarrow \texttt{true}}{G \vdash \mathcal{S}(F, (\texttt{if}(ce)\{s_1\} \ \texttt{else} \ \{s_2\}), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s_1, k, \sigma, E, M)}$$

$$\frac{G, \sigma, E, M \vdash ce \Rightarrow \texttt{false}}{G \vdash \mathcal{S}(F, (\texttt{if}(ce)\{s_1\} \ \texttt{else} \ \{s_2\}), k, \sigma, E, M) \xrightarrow{\epsilon} \mathcal{S}(F, s_2, k, \sigma, E, M)}$$

Fig. 9 Semantics of CminorSel. Only the rules that differ from those of Cminor are shown.

expression. For example, here are the smart constructor $\overline{\texttt{add}}$ for integer addition and its helper $\overline{\texttt{addi}}_n$ for immediate integer addition:

$$\begin{aligned}
\overline{\texttt{add}}(\texttt{addi}_{n_1}(a_1), \texttt{addi}_{n_2}(a_2)) &= \overline{\texttt{addi}}_{n_1+n_2}(\texttt{add}(a_1, a_2)) \\
\overline{\texttt{add}}(\texttt{addi}_n(a_1), a_2) &= \overline{\texttt{addi}}_n(\texttt{add}(a_1, a_2)) \\
\overline{\texttt{add}}(a_1, \texttt{addi}_n(a_2)) &= \overline{\texttt{addi}}_n(\texttt{add}(a_1, a_2)) \\
\overline{\texttt{add}}(n_1, n_2) &= n_1 + n_2 \\
\overline{\texttt{add}}(a_1', a_2') &= \texttt{add}(a_1', a_2') \text{ otherwise} \\
\overline{\texttt{addi}}_{n_1}(n_2) &= n_1 + n_2 \\
\overline{\texttt{addi}}_{n_1}(\texttt{addi}_{n_2}(a)) &= \texttt{addi}_{n_1+n_2}(a) \\
\overline{\texttt{addi}}_n(\texttt{addrsymbol}(id + \delta)) &= \texttt{addrsymbol}(id + (\delta + n)) \\
\overline{\texttt{addi}}_n(\texttt{addrstack}(\delta)) &= \texttt{addrstack}(\delta + n)
\end{aligned}$$

$$\overline{\texttt{addi}}_n(a) = \texttt{addi}_n(a) \text{ otherwise}$$

Here are some cases from other smart constructors that illustrate reassociation of immediate multiplication and immediate addition, as well as the recognition of the rotate-and-mask instruction:

$$\begin{aligned}
\overline{\texttt{muli}}_m(\texttt{addi}_n(a)) &= \overline{\texttt{addi}}_{m\times n}(\texttt{muli}_m(a)) \\
\overline{\texttt{shl}}(a,n) &= \overline{\texttt{rolm}}_{n,(-1)\ll n}(a) \\
\overline{\texttt{shru}}(a,n) &= \overline{\texttt{rolm}}_{32-n,(-1)\gg n}(a) \\
\overline{\texttt{and}}(a,n) &= \overline{\texttt{rolm}}_{0,n}(a) \\
\overline{\texttt{rolm}}_{n_1,m_1}(\texttt{rolm}_{n_2,m_2}(a)) &= \texttt{rolm}_{n_1+n_2,m}(a) \text{with } m = \texttt{rol}(m_1,n_2) \wedge m_2 \\
\overline{\texttt{or}}(\texttt{rolm}_{n,m_1}(a),\texttt{rolm}_{n,m_2}(a)) &= \texttt{rolm}_{n,m_1\vee m_2}(a)
\end{aligned}$$

While innocuous-looking, these smart constructors are powerful enough to, for instance, reduce $8 + (\texttt{x} + 1) \times 4$ to $\texttt{x} \times 4 + 12$, and to recognize a $\texttt{rolm}_{3,-1}(\texttt{x})$ instruction for `(x << 3) | (x >> 29)`, a C encoding of bit rotation commonly used in cryptography.

The recognition of conditions and addressing modes is performed by two functions $\overline{\texttt{cond}}(a) = c'$ and $\overline{\texttt{mode}}(a) = (mode, \vec{a})$. The translation of expressions is, then, a straightforward bottom-up traversal, applying the appropriate smart constructors at each step:

$$\begin{aligned}
[\![cst]\!] &= \overline{cst} \\
[\![op_1(a)]\!] &= \overline{op_1}([\![a]\!]) \\
[\![op_2(a_1,a_2)]\!] &= \overline{op_2}([\![a_1]\!],[\![a_2]\!]) \\
[\![\kappa[a]]\!] &= \texttt{load}(\kappa, mode, \vec{a}) \text{ where } (mode,\vec{a}) = \overline{\texttt{mode}}([\![a]\!]) \\
[\![a_1 \ ?\ a_2 : a_3]\!] &= \overline{\texttt{cond}}([\![a_1]\!])\ ?\ [\![a_2]\!] : [\![a_3]\!]
\end{aligned}$$

We omit the translation of statements and functions, which is similar.

5.3 Semantic preservation

The first part of the proof that instruction selection preserves semantics is to show the correctness of the smart constructor functions.

Lemma 2

1. *If* $G,\sigma,E,M \vdash a_1 \Rightarrow v_1$ *and* $\texttt{eval\_unop}(op_1,v_1) = \lfloor v \rfloor$, *then* $G,\sigma,E,M \vdash \overline{op_1}(a_1) \Rightarrow v$.
2. *If* $G,\sigma,E,M \vdash a_1 \Rightarrow v_1$ *and* $G,\sigma,E,M \vdash a_2 \Rightarrow v_2$ *and* $\texttt{eval\_binop}(op_2,v_1,v_2) = \lfloor v \rfloor$, *then* $G,\sigma,E,M \vdash \overline{op_2}(a_1,a_2) \Rightarrow v$.
3. *If* $G,\sigma,E,M \vdash a \Rightarrow v$ *and* $\texttt{istrue}(v)$, *then* $G,\sigma,E,M \vdash \overline{\texttt{cond}}(a) \Rightarrow \texttt{true}$.
4. *If* $G,\sigma,E,M \vdash a \Rightarrow v$ *and* $\texttt{isfalse}(v)$, *then* $G,\sigma,E,M \vdash \overline{\texttt{cond}}(a) \Rightarrow \texttt{false}$.
5. *If* $G,\sigma,E,M \vdash a \Rightarrow v$ *and* $\overline{\texttt{mode}}(a) = (mode,\vec{a})$. *then there exists* $\vec{v}$ *such that* $G,\sigma,E,M \vdash \vec{a} \Rightarrow \vec{v}$ *and* $\texttt{eval\_mode}(mode,\vec{v}) = \lfloor v \rfloor$.

After copious case analysis on the operators and their arguments and inversion on the evaluations of the arguments, the proof reduces to showing that the defining

equations for the smart constructors are valid when interpreted over ground machine integers. For instance, in the case for $\overline{\texttt{rolm}}$ shown above, we have to prove that

$$\texttt{rol}((\texttt{rol}(x, n_1) \wedge m_1), n_2) \wedge m_2 \;=\; \texttt{rol}(x, n_1 + n_2) \wedge (\texttt{rol}(m_1, n_2) \wedge m_2)$$

which follows from the algebraic properties of rotate-and-left and bitwise "and". Completing the proof of the lemma above required the development of a rather large and difficult formalization of N-bit machine integers and of the algebraic properties of their arithmetic and logical operations.

Let P be the original Cminor program and P' be the CminorSel program produced by instruction selection. Let G, G' be the corresponding global environments. Semantic preservation for the evaluation of expressions follows from lemma 2 by induction on the Cminor evaluation derivation.

Lemma 3 *If $G, \sigma, E, M \vdash a \Rightarrow v$, then $G', \sigma, E, M \vdash [\![a]\!] \Rightarrow v$.*

The last part of the semantic preservation proof is a simulation argument of the form outlined in section 3.7. Since the structure of statements is preserved by the translation, transitions in the original and transformed programs match one-to-one, resulting in a "lock-step" simulation diagram. The relation $\sim$ between Cminor and CminorSel execution states is defined as follows:

$$\begin{aligned} \mathcal{S}(F, s, k, \sigma, E, M) &\sim \mathcal{S}([\![F]\!], [\![s]\!], [\![k]\!], \sigma, E, M) \\ \mathcal{C}(Fd, \vec{v}, k, M) &\sim \mathcal{C}([\![Fd]\!], \vec{v}, [\![k]\!], M) \\ \mathcal{R}(v, k, M) &\sim \mathcal{R}(v, [\![k]\!], M) \end{aligned}$$

Since the transformed code computes exactly the same values as the original, environments E and memory states M are identical in matching states. Statements and functions appearing in states must be the translation of one another. For continuations, we extend (isomorphically) the translation of statements and functions.

Lemma 4 *If $G \vdash S_1 \xrightarrow{t} S_2$ and $S_1 \sim S_1'$, there exists S_2' such that $G' \vdash S_1' \xrightarrow{t} S_2'$ and $S_1' \sim S_2'$.*

The proof is a straightforward case analysis on the transition from S_1 to S_2. Semantic preservation for instruction selection then follows from theorem 3 and lemma 4.

6 RTL generation

The second compilation pass translates CminorSel to a simple intermediate language of the RTL kind, with control represented as a control-flow graph instead of structured statements. This intermediate language is convenient for performing optimizations later.

6.1 The target language: RTL

The RTL language represents functions as a control-flow graph (CFG) of abstract instructions, corresponding roughly to machine instructions but operating over pseudo-registers (also called "temporaries"). Every function has an unlimited supply of pseudo-registers, and their values are preserved across function call. In the following, r ranges over pseudo-registers and l over labels of CFG nodes.

RTL instructions:	$i ::= \texttt{nop}(l)$	no operation (go to l)
	$\mid \texttt{op}(op, \vec{r}, r, l)$	arithmetic operation
	$\mid \texttt{load}(\kappa, mode, \vec{r}, r, l)$	memory load
	$\mid \texttt{store}(\kappa, mode, \vec{r}, r, l)$	memory store
	$\mid \texttt{call}(sig, (r \mid id), \vec{r}, r, l)$	function call
	$\mid \texttt{tailcall}(sig, (r \mid id), \vec{r})$	function tail call
	$\mid \texttt{cond}(cond, \vec{r}, l_{true}, l_{false})$	conditional branch
	$\mid \texttt{return} \mid \texttt{return}(r)$	function return
RTL control-flow graph:	$g ::= l \mapsto i$	finite map
RTL functions:	$F ::= \{\ \texttt{sig} = sig;$	
	$\quad \texttt{params} = \vec{r};$	parameters
	$\quad \texttt{stacksize} = n;$	size of stack data block
	$\quad \texttt{entrypoint} = l;$	label of first instruction
	$\quad \texttt{code} = g\}$	control-flow graph

Each instruction takes its arguments in a list of pseudo-registers $\vec{r}$ and stores its result, if any, in a pseudo-register r. Additionally, it carries the labels of its possible successors. We use instructions rather than basic blocks as nodes of the control-flow graph because this simplifies semantics and reasoning over static analyses without significantly slowing compilation [50].

The dynamic semantics of RTL is defined by the labeled transition system shown in figure 10. Program states have the following form:

Program states:	$S ::= \mathcal{S}(\Sigma, g, \sigma, l, R, M)$	regular state
	$\mid \mathcal{C}(\Sigma, Fd, \vec{v}, M)$	call state
	$\mid \mathcal{R}(\Sigma, v, M)$	return state
Call stacks:	$\Sigma ::= (\mathcal{F}(r, F, \sigma, l, R))^*$	list of frames
Register states:	$R ::= r \mapsto v$	

In regular states, g is the CFG of the function currently executing, l a program point (CFG node label) within this function, σ its stack data block, and R an assignment of values for the pseudo-registers of F. All three states carry a call stack Σ, which is a list of frames $\mathcal{F}$ representing pending function calls and containing the corresponding per-function state F, σ, l, R.

The transition system in figure 10 is unsurprising. Transitions from a regular state discriminate on the instruction found at the current program point. To interpret arithmetic operations, conditions and addressing modes, we reuse the functions `eval_op`, `eval_cond` and `eval_mode` of the CminorSel semantics. Other transitions follow the pattern described in section 3.6.

6.2 Relational specification of the translation

The translation from CminorSel to RTL is conceptually simple: the structured control is encoded as a CFG[4]; expressions are decomposed into sequences of RTL instructions;

[4] Since RTL currently has no instructions performing N-way branches (i.e. jump tables), this translation of control includes the generation of binary decision trees for Cminor `switch` statements. We do not describe this part of the translation in this article.

$$\frac{g(l) = \lfloor \texttt{nop}(l') \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l', R, M)}$$

$$\frac{g(l) = \lfloor \texttt{op}(op, \vec{r}, r, l') \rfloor \quad \texttt{eval\_op}(G, \sigma, op, R(\vec{r})) = \lfloor v \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l', R\{r \leftarrow v\}, M)}$$

$$\frac{\begin{array}{c} g(l) = \lfloor \texttt{load}(\kappa, mode, \vec{r}, r, l') \rfloor \\ \texttt{eval\_mode}(G, \sigma, mode, R(\vec{r})) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{load}(M, \kappa, b, \delta) = \lfloor v \rfloor \end{array}}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l', R\{r \leftarrow v\}, M)}$$

$$\frac{\begin{array}{c} g(l) = \lfloor \texttt{store}(\kappa, mode, \vec{r}, r, l') \rfloor \\ \texttt{eval\_mode}(G, \sigma, mode, R(\vec{r})) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{store}(M, \kappa, b, \delta, R(r)) = \lfloor M' \rfloor \end{array}}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l', R, M')}$$

$$\frac{g(l) = \lfloor \texttt{call}(sig, r_f, \vec{r}, r, l') \rfloor \quad R(r_f) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{C}(\mathcal{F}(r, g, \sigma, l', R).\Sigma, Fd, R(\vec{r}), M)}$$

$$\frac{g(l) = \lfloor \texttt{tailcall}(sig, r_f, \vec{r}) \rfloor \quad R(r_f) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{C}(\Sigma, Fd, R(\vec{r}), \texttt{free}(M, \sigma))}$$

$$\frac{g(l) = \lfloor \texttt{cond}(cond, \vec{r}, l_{true}, l_{false}) \rfloor \quad \texttt{eval\_cond}(cond, R(\vec{r})) = \lfloor \texttt{true} \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l_{true}, R, M)}$$

$$\frac{g(l) = \lfloor \texttt{cond}(cond, \vec{r}, l_{true}, l_{false}) \rfloor \quad \texttt{eval\_cond}(cond, R(\vec{r})) = \lfloor \texttt{false} \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l_{false}, R, M)}$$

$$\frac{g(l) = \lfloor \texttt{return} \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{R}(\Sigma, \texttt{undef}, \texttt{free}(M, \sigma))}$$

$$\frac{g(l) = \lfloor \texttt{return}(r) \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\epsilon} \mathcal{R}(\Sigma, R(r), \texttt{free}(M, \sigma))}$$

$$\frac{\texttt{alloc}(M, 0, F.\texttt{stacksize}) = (\sigma, M')}{G \vdash \mathcal{C}(\Sigma, \texttt{internal}(F), \vec{v}, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F.\texttt{code}, \sigma, F.\texttt{entrypoint}, [F.\texttt{params} \mapsto \vec{v}], M')}$$

$$\frac{\vdash Fe(\vec{v}) \xRightarrow{t} v \text{ (see section 3.4)}}{G \vdash \mathcal{C}(\Sigma, \texttt{external}(Fe), \vec{v}, M) \xrightarrow{t} \mathcal{R}(\Sigma, v, M)}$$

$$G \vdash \mathcal{R}(\mathcal{F}(r, g, \sigma, l, R).\Sigma, v, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, g, \sigma, l, R[r \leftarrow v], M)$$

$$\frac{\texttt{symbol}(\texttt{globalenv}(P), P.\texttt{main}) = \lfloor b \rfloor \quad \texttt{funct}(\texttt{globalenv}(P), b) = \lfloor Fd \rfloor}{\texttt{initial}(P, \mathcal{C}(\epsilon, Fd, \epsilon, \texttt{initmem}(P)))}$$

$$\texttt{final}(\mathcal{R}(\epsilon, \texttt{int}(n), M), n)$$

Fig. 10 Semantics of RTL.

pseudo-registers are generated to hold the values of CminorSel variables and intermediate results of expression evaluations. The decomposition of expressions is made trivial by the prior conversion to CminorSel: every operation becomes exactly one op instruction. However, implementing this translation in Coq is delicate: since Coq is a pure functional language, we cannot use imperative updates to build the CFG and generate fresh pseudo-registers and CFG nodes. Section 6.4 describes a solution, based (unsurprisingly) on the use of a monad.

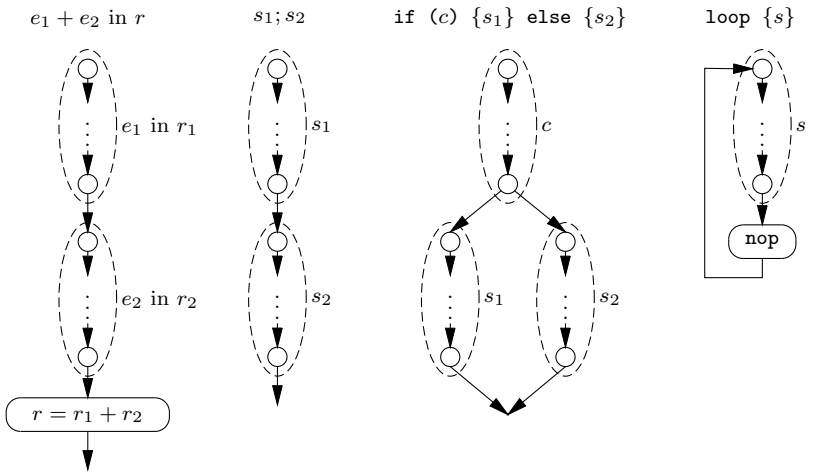

Fig. 11 Examples of correspondences between expressions/statements and sub-graphs of a CFG.

In the present section, we give a relational, non-executable specification of the translation: syntactic conditions under which a RTL function is an acceptable translation of a CminorSel function. The key intuition captured by this specification is that each subexpression or substatement of the CminorSel function should correspond to a sub-graph of the RTL control-flow graph. For an expression a, this sub-graph is identified by a start node l_1 and an end node l_e. The instructions on the paths from l_1 to l_e should compute the value of expression a, deposit it in a given destination register r_d, and preserve the values of a given set of registers. For a statement s, the sub-graph has one start node but several end nodes corresponding to the multiple ways in which a statement can terminate (normally, by `exit`, or by `goto`). Figure 11 illustrates this correspondence. As depicted there, the sub-graph for a compound expression or compound statement such as $s_1; s_2$ contains sub-graphs for its components s_1 and s_2, suitably connected. In other words, the relational specification of the translation describes a hierarchical decomposition of the CFG along the structure of statements and expressions of the original CminorSel code.

The specification for expressions is the predicate $g, \gamma, \pi \vdash a \texttt{ in } r_d \sim l_1, l_2$, where g is the CFG, γ an injective mapping from CminorSel variables to the registers holding their values, π a set of registers that the instructions evaluating a must preserve (in addition to those in $\mathrm{Rng}(\gamma)$), a the CminorSel expression, r_d the register where its value must be deposited, and l_1, l_2 the start and end nodes in the CFG. The following rules give the flavor of the specification:

$$\frac{\gamma(id) = \lfloor r_d \rfloor}{g, \gamma, \pi \vdash id \texttt{ in } r_d \sim l_1, l_1}$$

$$\frac{\gamma(id) = \lfloor r \rfloor \qquad g(l_1) = \lfloor \mathtt{op}(\mathtt{move}, r, r_d, l_2) \rfloor \qquad r_d \notin \mathrm{Rng}(\gamma) \cup \pi}{g, \gamma, \pi \vdash id \texttt{ in } r_d \sim l_1, l_2}$$

$$\frac{g,\gamma,\pi \vdash \vec{a}\ \texttt{in}\ \vec{r} \sim l_1, l \qquad g(l) = \lfloor \texttt{op}(op, \vec{r}, r_d, l_2) \rfloor \qquad r_d \notin \mathrm{Rng}(\gamma) \cup \pi}{g,\gamma,\pi \vdash op(\vec{a})\ \texttt{in}\ r_d \sim l_1, l_2}$$

$$g,\gamma,\pi \vdash \epsilon\ \texttt{in}\ \epsilon \sim l_1, l_1$$

$$\frac{g,\gamma,\pi \vdash a\ \texttt{in}\ r \sim l_1, l \qquad g,\gamma,\pi \cup \{r\} \vdash \vec{a}\ \texttt{in}\ \vec{r} \sim l, l_2 \qquad r \notin \mathrm{Rng}(\gamma) \cup \pi}{g,\gamma,\pi \vdash a.\vec{a}\ \texttt{in}\ r.\vec{r} \sim l_1, l_2}$$

The freshness side-conditions $r \notin \mathrm{Rng}(\gamma) \cup \pi$ ensure that the temporary registers used to hold the values of subexpressions of a do not interfere with registers holding values of CminorSel variables ($\mathrm{Rng}(\gamma)$) nor with temporary registers holding values of previously computed subexpressions (e.g. in the expression $\texttt{add}(a_1, a_2)$, the value of a_1 during the computation of the value of a_2). The specification for conditional expressions is similar: $g,\gamma,\pi \vdash c \sim l_1, l_{true}, l_{false}$, but with two exit nodes, l_{true} and l_{false}. The instruction sequence starting at l_1 should terminate at l_{true} if c evaluates to `true` and at l_{false} if c evaluates to `false`. Finally, the translation of statements is specified by the predicate $g,\gamma \vdash s \sim l_1, l_2, l_e, l_g, l_r, r_r^?$, where l_e is a list of nodes and l_g a mapping from CminorSel labels to nodes. The contract expressed by this complicated predicate is that the instruction sequence starting at l_1 should compute whatever s computes and branch to node l_2 if s terminates normally, to $l_e(n)$ if s terminates by $\texttt{exit}(n)$, to $l_g(lbl)$ if s performs `goto` lbl, and to l_r if s performs a `return` (after depositing the return value, if any, in register $r_r^?$). For simplicity, figure 11 depicts only the l_2 final node, and the following sample rules consider only the l_2 and l_e final nodes.

$$\frac{\gamma(id) = \lfloor r_d \rfloor \qquad g,\gamma,\emptyset \vdash a\ \texttt{in}\ r_d \sim l_1, l_2}{g,\gamma \vdash (id = a) \sim l_1, l_2, l_e}$$

$$\frac{\gamma(id) = \lfloor r \rfloor \qquad g,\gamma,\emptyset \vdash a\ \texttt{in}\ r_d \sim l_1, l \qquad g(l) = \lfloor \texttt{op}(\texttt{move}, r_d, r, l_2) \rfloor}{g,\gamma \vdash (id = a) \sim l_1, l_2, l_e}$$

$$\frac{g,\gamma \vdash s_1 \sim l_1, l, l_e \qquad g,\gamma \vdash s_2 \sim l, l_2, l_e}{g,\gamma \vdash (s_1; s_2) \sim l_1, l_2, l_e}$$

$$\frac{\begin{array}{c} g,\gamma,\emptyset \vdash c \sim l_1, l_{true}, l_{false} \\ g,\gamma \vdash s_1 \sim l_{true}, l_2, l_e \qquad g,\gamma \vdash s_2 \sim l_{false}, l_2, l_e \end{array}}{g,\gamma \vdash \texttt{if}(c)\{s_1\}\ \texttt{else}\ \{s_2\} \sim l_1, l_2, l_e}$$

$$\frac{g(l_1) = \lfloor \texttt{nop}(l) \rfloor \qquad g,\gamma \vdash s \sim l, l_1, l_e}{g,\gamma \vdash \texttt{loop}\{s\} \sim l_1, l_2, l_e} \qquad \frac{g,\gamma \vdash s \sim l_1, l_2, l_2.l_e}{g,\gamma \vdash \texttt{block}\{s\} \sim l_1, l_2, l_e}$$

$$\frac{l_e(n) = \lfloor l_1 \rfloor}{g,\gamma \vdash \texttt{exit}(n) \sim l_1, l_2, l_e}$$

The specification for the translation of CminorSel functions to RTL functions is, then: if $[\![F]\!] = \lfloor F' \rfloor$, it must be the case that

$$F'.\texttt{code}, \gamma \vdash F.\texttt{body} \sim F'.\texttt{entrypoint}, l, \epsilon, l_g, l, r_r^?$$

for some l, l_g and injective γ; moreover, the CFG node at l must contain a $\texttt{return}(r_r^?)$ instruction appropriate to the signature of F.

6.3 Semantic preservation

We now outline the proof of semantic preservation for RTL generation. Consider a CminorSel program P and an RTL program P', with G, G' being the corresponding global environments. Assuming that P' is an acceptable translation of P according to the relational specification of section 6.2, we show that executions of P are simulated by executions of P'. For the evaluation of CminorSel expressions, the simulation argument is of the form "if expression a evaluates to value v, the generated RTL code performs a sequence of transitions from l_1 to l_2, where l_1, l_2 delimit the sub-graph of the CFG corresponding to a, and leaves v in the given destination register r_d". Agreement between a CminorSel environment E and an RTL register state R is written $\gamma \vdash E \sim R$ and defined as $R(\gamma(x)) = E(x)$ for all $x \in \mathrm{Dom}(\gamma)$.

Lemma 5 *Assume* $F.\texttt{code}, \gamma, \pi \vdash a$ `in` $r_d \sim l_1, l_2$ *and* γ *is injective.*
If $G, \sigma, E, M \vdash a \Rightarrow v$ *and* $\gamma \vdash E \sim R$*, there exists* R' *such that*

1. *The* RTL *code executes from* l_1 *to* l_2*:*

$$G' \vdash \mathcal{S}(\Sigma, F, \sigma, l_1, R, M) \xrightarrow{\epsilon}{}^{*} \mathcal{S}(\Sigma, F, \sigma, l_2, R', M)$$

2. *Register* r_d *contains the value of* v *at the end of this execution:* $R'(r_d) = v$
3. *The values of preserved registers are unchanged:* $R'(r) = R(r)$ *for all* $r \in \mathrm{Rng}(\gamma) \cup \pi$*. This implies* $\gamma \vdash E \sim R'$ *in particular.*

This lemma, along with a similar lemma for condition expressions, is proved by induction on the CminorSel evaluation derivation. To relate CminorSel and RTL execution states, we need to define a correspondence between CminorSel continuations and RTL call stacks. A continuation k interleaves two aspects that are handled separately in RTL. The parts of k that lie between `returnto` markers, namely the continuations $s; k'$ and $\texttt{endblock}(k')$, correspond to execution paths within the current function. These paths connect the several possible end points for the current statement with the final `return` of the function. Just as a statement s is associated with a "fan-out" subgraph of the CFG (one start point, several end points), this part of k is associated with a "fan-in" subgraph of the CFG (several start points, one end point corresponding to a `return` instruction). The other parts of k, namely the `returnto` markers, are in one-to-one correspondence with frames on the RTL call stack Σ. We formalize these intuitions using two mutually inductive predicates: $g, \gamma \vdash k \sim l_2, l_e, l_g, l_r, r_r^?, \Sigma$ for the local part of the continuation k, and $k \sim S$ for call continuations k. The definitions are omitted for brevity. The invariant between CminorSel states and RTL states is, then, of the following form.

$$\frac{g, \gamma \vdash s \sim l_1, l_2, l_e, l_g, l_r, r_r^? \qquad g, \gamma \vdash k \sim l_2, l_e, l_g, l_r, r_r^?, \Sigma \qquad \gamma \vdash E \sim R}{\mathcal{S}(F, s, k, \sigma, E, M) \sim \mathcal{S}(\Sigma, g, \sigma, l_1, R, M)}$$

$$\frac{[\![Fd]\!] = \lfloor Fd' \rfloor \qquad k \sim \Sigma}{\mathcal{C}(Fd, \vec{v}, k, M) \sim \mathcal{C}(\Sigma, Fd', \vec{v}, M)} \qquad \frac{k \sim \Sigma}{\mathcal{R}(v, k, M) \sim \mathcal{R}(\Sigma, v, M)}$$

The proof of semantic preservation for statements and functions is a simulation diagram of the "star" kind (see section 3.7). Several CminorSel transitions become no-operations in the translated RTL code, such as self assignments $id = id$ and $\texttt{exit}(n)$

constructs. We therefore need to define a measure over CminorSel states that decreases on such potentially stuttering steps. After trial and error, an appropriate measure for regular states is the lexicographically-ordered pair of nonnegative integers $|\mathcal{S}(F, s, k, \sigma, E, M)| = (|s| + |k|, |s|)$ where $|s|$ is the number of nodes in the abstract syntax tree for s, and

$$|s; k| = 1 + |s| + |k| \qquad |\texttt{endblock}(k)| = 1 + |k| \qquad |k| = 0 \text{ otherwise.}$$

The measure for call states and return states is $(0, 0)$.

Lemma 6 *If $G \vdash S_1 \xrightarrow{t} S_2$ and $S_1 \sim S_1'$, either there exists S_2' such that $G' \vdash S_1' \xrightarrow{t}{}^{+} S_2'$ and $S_1' \sim S_2'$, or $|S_2| < |S_1|$ and there exists S_2' such that $G' \vdash S_1' \xrightarrow{t}{}^{*} S_2'$ and $S_1' \sim S_2'$.*

The proof is a long case analysis on the transition $G \vdash S_1 \xrightarrow{t} S_2$. Semantic preservation for RTL generation then follows from theorem 4.

6.4 Functional implementation of the translation

We now return to the question left open at the beginning of section 6.3: how to define the generation of RTL as a Coq function? Naturally, the translation proceeds by a recursive traversal of CminorSel expressions and statements, incrementally adding the corresponding instructions to the CFG and generating fresh temporary registers to hold intermediate results within expressions. Additionally, the translation may fail, e.g. if an undeclared local variable is referenced or a statement label is defined several times. This would cause no programming difficulties in a language featuring mutation and exceptions, but these luxuries are not available in Coq, which is a pure functional language. We therefore use a monadic programming style using the state-and-error monad. The compile-time state is a triple (g, l, r) where g is the current state of the CFG, l the next unused CFG node label, and r the next unused pseudo-register. Every translation that computes (imperatively) a result of type α becomes a pure function with type $\texttt{mon}(\alpha) = \texttt{state} \to \texttt{Error} \mid \texttt{OK}(\texttt{state} \times \alpha)$. Besides the familiar `ret` and `bind` monadic combinators, the basic operations of this monad are:

- `newreg` : `mon`(`reg`) generates a fresh temporary register (by incrementing the r component of the state);
- `add_instr`(i) : `mon`(`node`) allocates a fresh CFG node l, stores the instruction i in this node, and returns l;
- `reserve_instr` : `mon`(`node`) allocates and returns a fresh CFG node, leaving it empty;
- `update_instr`(l, i) : `mon`(`unit`) stores instruction i in node l, raising an error if node l is not empty.

(The latter two operations are used when compiling loops and labeled statements.) The translation functions, then, are of the following form:

$$\begin{array}{ll}
\texttt{transl_expr}(\gamma, a, r_d, l_2) & : \texttt{mon}(\texttt{node}) \\
\texttt{transl_exprlist}(\gamma, \vec{a}, \vec{r_d}, l_2) & : \texttt{mon}(\texttt{node}) \\
\texttt{transl_condition}(\gamma, c, l_{true}, l_{false}) & : \texttt{mon}(\texttt{node}) \\
\texttt{transl_stmt}(\gamma, s, l_2, l_e, l_g, l_r, r_r^?) & : \texttt{mon}(\texttt{node})
\end{array}$$

These functions recursively add to the CFG the instructions that compute the given expression or statement and branch to the given end nodes $l_2, l_e, \ldots$. Each call returns the node of the first instruction in this sequence (the node written l_1 in the relational specification). The following Coq excerpts from `transl_stmt` should give the flavor of the translation functions:

```
match s with
| Sskip =>
    ret l2
| Sassign v a =>
    do rd <- new_reg;
    do rv <- find_var γ v;
    do l  <- add_instr (Iop Omove (rd::nil) rv l2);
    transl_expr γ a rd l
| Sseq s1 s2 =>
    do l <- transl_stmt γ s2 l2 lexit lgoto lret rret;
    transl_stmt γ s1 l lexit lgoto lret rret
| Sloop s =>
    do l <- reserve_instr;
    do l' <- transl_stmt γ s l lexit lgoto lret rret;
    do x  <- update_instr l (Inop l');
    ret l
```

Inspired by Haskell, `do` x `<-` $a; b$ is a user-defined Coq notation standing for `bind` a $(\lambda x.b)$. Two syntactic invariants of the state play a crucial role in proving the correctness of the generated CFG against the relational specification of section 6.2. First, in a compile-time state (g, l, r), all CFG nodes above l must be empty: $\forall l' \geq l$, $g(l') = \emptyset$. Second, the state evolves in a monotone way: nodes are only added to the CFG, but an already filled node is never modified; likewise, temporary registers are never reused. (If this were not the case, correct sub-graphs constructed by recursive invocations to the `transl` functions could become incorrect after later modifications of the CFG.) We define this monotonicity property as a partial order $\preceq$ between states:

$$(g_1, l_1, r_1) \preceq (g_2, l_2, r_2) \quad \stackrel{\text{def}}{=} \quad l_1 \leq l_2 \wedge r_1 \leq r_2 \wedge (\forall l, i,\ g_1(l) = \lfloor i \rfloor \Rightarrow g_2(l) = \lfloor i \rfloor)$$

It is straightforward but tedious to show that these two invariants are satisfied by the monadic translation functions, since they hold for the basic operations of the monad and are preserved by monadic `bind` composition. However, we can avoid much proof effort by taking advantage of Coq's dependent types. The first invariant (of one state) can be made a part of the state itself, which becomes a dependent record type:

```
Record state: Set := mkstate {
  st_nextreg: reg;
  st_nextnode: node;
  st_code: graph;
  st_wf: forall (l: node), l >= st_nextnode -> st_code!l = None
}.
```

Note the 4^{th} field `st_wf`, which is a proof term that the first invariant holds. The second invariant (the partial order $\preceq$) is more difficult, as it involves two states. However, it can be expressed in the definition of the `mon`(α) type by turning the function type

`state` $\to \ldots$ into a dependent function type of the shape $\Pi(x : \texttt{state}).\ \{y \mid P\ x\ y\}$. Here is the Coq definition of the dependently-typed monad:

```
Inductive res (A: Set) (s: state): Set :=
  | Error: res A s
  | OK: A -> forall (s': state), s ⪯ s' -> res A s.
Definition mon (A: Set) : Set := forall (s: state), res A s.
```

The result of a successful monadic computation of type $\texttt{mon}(\alpha)$ starting in state s is $\texttt{OK}(x, s', \pi)$ where $x : \alpha$ is the return value, s' the final state, and π a proof term for the proposition $s \preceq s'$. The two invariants need to be proved when defining the basic operations of the monad; for instance, in the case of `ret` and `bind`, the corresponding proofs amount to reflexivity and transitivity of $\preceq$. However, they then automatically hold for all computations in the dependently-typed monad.

7 Optimizations based on dataflow analysis

We now describe two optimization passes performed on the RTL form: constant propagation and common subexpression elimination. Both passes make use of a generic solver for dataflow inequations, which we describe first.

7.1 Generic solvers for dataflow inequations

We formalize forward dataflow analyses as follows. We are given a control-flow graph (as a function $\texttt{successors} : \texttt{node} \to \texttt{list}(\texttt{node})$) and a transfer function $T : \texttt{node} \times \mathcal{A} \to \mathcal{A}$, where $\mathcal{A}$ is the type of abstract values (the results of the analysis), equipped with a partial order $\geq$. Intuitively, $T(l)$ computes the abstract value "after" the instruction at point l as a function of the abstract value "before" this instruction. We are also given a set `cstrs` of pairs of a CFG node and an abstract value $a : \mathcal{A}$, representing e.g. requirements on the CFG entry point. The result of forward dataflow analysis is a solution $A : \texttt{node} \to \mathcal{A}$ to the following forward dataflow inequations:

$$A(s) \geq T(l, A(l)) \text{ for all } s \in \texttt{successors}(l)$$
$$A(l) \geq a \text{ for all } (l, a) \in \texttt{cstrs}$$

We formalize dataflow analysis as inequations instead of the usual equations $A(l) = \bigsqcup \{T(p, A(p)) \mid l \in \texttt{successors}(p)\}$ because we are interested only in the correctness of the solutions, not in their optimality.

Two solvers for dataflow inequations are provided as Coq functors, that is, modules parameterized by a module defining the type $\mathcal{A}$ and its operations. The first solver implements Kildall's worklist algorithm [48]. It is applicable if the type $\mathcal{A}$ is equipped with a decidable equality, a least element $\bot$ and an upper bound operation $\sqcup$. (Again, since we are not interested in optimality of the results, $\sqcup$ is not required to compute the least upper bound.) The second solver performs propagation over extended basic blocks, setting $A(l) = \top$ in the solution for all points l that have several predecessors. The only requirement over the type $\mathcal{A}$ is that it possesses a greatest element $\top$. This propagation-based solver is useful in cases where upper bounds do not always exist or are too expensive to compute.

Several mechanical verifications of Kildall's worklist algorithm have been published already [5, 26, 49, 12]. Therefore, we do not detail the correctness proofs for our solvers, referring the reader to the cited papers and to the Coq development for more details.

The solvers actually return an option type, with $\emptyset$ denoting failure and $\lfloor A \rfloor$ denoting success with solution A. For simplicity, we do not require that the CFG is finite, nor in the case of Kildall's algorithm that the $\geq$ ordering over A is well founded (no infinite ascending chains). Consequently, we cannot guarantee termination of the solvers and must allow them to fail, at least formally. The implementations of the two solvers bound the number of iterations performed to find a fixed point, returning $\emptyset$ if a solution cannot be found in N iterations, where N is a very large constant. Alternatively, unbounded iteration can be implemented using the approach of Bertot and Komendantsky [13], which uses classical logic and Tarski's theorem to model general recursion in Coq. Yet another alternative is to use the "verified validator" approach, where the computation of the solution is delegated to external, untrusted code, then verified a posteriori to satisfy the dataflow inequations. In all these approaches, if the static analysis fails, we can either abort the compilation process or simply turn off the corresponding optimization pass, returning the input code unchanged.

Solvers for backward dataflow inequations can be easily derived from the forward solvers by reversing the edges of the control-flow graph. In the backward case, the transfer function $T(l)$ computes the abstract value "before" the instruction at point l as a function of the abstract value "after" this instruction. The solution A returned by the solvers satisfies the backward dataflow inequations:

$$A(l) \geq T(s, A(s)) \text{ for all } s \in \texttt{successors}(l)$$
$$A(l) \geq a \text{ for all } (l, a) \in \texttt{cstrs}$$

7.2 Constant propagation

7.2.1 Static analysis

Constant propagation for a given function starts with a forward dataflow analysis using the following domain of abstract values:

$$\mathcal{A} \stackrel{\text{def}}{=} r \mapsto (\top \mid \bot \mid \texttt{Int}(n) \mid \texttt{Float}(f) \mid \texttt{Addrsymbol}(id + \delta)$$

That is, at each program point and for each register r, we record whether its value at this point is statically known to be equal to an integer n, a float f, or the address of a symbol id plus an offset δ, or is unknown ($\top$), or whether this program point is unreachable ($\bot$). Kildall's algorithm is used to solve the dataflow inequations, with the additional constraint that $A(F.\texttt{entrypoint}) \geq (r \mapsto \top)$. If it fails to find a proper solution A, we take the trivial solution $A(l) = (r \mapsto \top)$, effectively turning off the optimization. For each function F of the program, we write $\texttt{analyze}(F)$ for the solution (proper or trivial) of the dataflow equations for F.

The transfer function T_F is the obvious abstract interpretation of RTL's semantics on this domain:

$$T_F(l, a) = \begin{cases} a\{r \leftarrow \overline{\texttt{eval\_op}}(op, a(\vec{r}))\} & \text{if } F.\texttt{code}(l) = \lfloor \texttt{op}(op, \vec{r}, r, l') \rfloor \\ a\{r \leftarrow \top\} & \text{if } F.\texttt{code}(l) = \lfloor \texttt{load}(\kappa, mode, \vec{r}, r, l') \rfloor \\ a\{r \leftarrow \top\} & \text{if } F.\texttt{code}(l) = \lfloor \texttt{call}(sig, \_, r, l') \rfloor \\ a & \text{otherwise} \end{cases}$$

Here, $\overline{\texttt{eval\_op}}$ is the abstract interpretation over the domain $\mathcal{A}$ of the `eval_op` function defining the semantics of operators. By lack of an alias analysis, we do not attempt to track constant values stored in memory; therefore, the abstract value of the result of a `load` is $\top$.

7.2.2 Code transformation

The code transformation exploiting the results of this analysis is straightforward: `op` instructions become "load constant" instructions if the values of all argument registers are statically known; `cond` instructions where the condition can be statically determined to be always `true` or always `false` are turned into `nop` instructions branching to the appropriate successor; finally, operators, conditions and addressing modes are specialized to cheaper immediate forms if the values of some of their arguments are statically known. The structure of the control-flow graph is preserved (no node is inserted nor deleted), making this transformation easy to express as a morphism over the CFG. Parts of the CFG can become unreachable as a consequence of statically resolving some `cond` instructions. The corresponding instructions, as well as the `nop` instructions that replace the statically-resolved `cond` instructions, will be removed later during branch tunneling and CFG linearization (sections 9 and 10).

7.2.3 Semantic preservation

The proof of semantic preservation for constant propagation is based on a "lock-step" simulation diagram. The central invariant of the diagram is the following: at every program point l within a function F, the concrete values $R(r)$ of registers r must agree with the abstract values $\texttt{analyze}(F)(l)(r)$ predicted by the dataflow analysis. Agreement between a concrete and an abstract value is written $\models a : v$ and defined as follows.

$$\models v : \top \qquad \models \texttt{int}(n) : \texttt{Int}(n) \qquad \models \texttt{float}(f) : \texttt{Float}(f)$$

$$\frac{\texttt{symbol}(G, id) = \lfloor b \rfloor}{\models \texttt{ptr}(b, \delta) : \texttt{Addrsymbol}(id + \delta)}$$

We write $\models R : A$ to mean $\models R(r) : A(r)$ for all registers r. The first part of the proof shows the correctness of the abstract interpretation with respect to this agreement relation. For example, if $\texttt{eval\_op}(op, \vec{v}) = \lfloor v \rfloor$ and $\models \vec{v} : \vec{a}$, we show that $\models v : \overline{\texttt{eval\_op}}(op, \vec{a})$. Likewise, we show that the specialized forms of operations, addressing modes and conditions produced by the code transformation compute the same values as the original forms, provided the concrete arguments agree with the abstract values used to determine the specialized forms. These proofs are large but straightforward case analyses. We then define the relation between pairs of RTL states that is invariant under execution steps.

$$\frac{[\![F]\!] = \lfloor F' \rfloor \qquad \Sigma \sim \Sigma' \qquad \models R : \texttt{analyze}(F)(l)}{\mathcal{S}(\Sigma, F.\texttt{code}, \sigma, l, R, M) \sim \mathcal{S}(\Sigma', F'.\texttt{code}, \sigma, l, R, M)}$$

$$\frac{[\![Fd]\!] = \lfloor Fd' \rfloor \qquad \Sigma \sim \Sigma'}{\mathcal{C}(\Sigma, Fd, \vec{v}, M) \sim \mathcal{C}(\Sigma', Fd', \vec{v}, M)} \qquad \frac{\Sigma \sim \Sigma'}{\mathcal{C}(\Sigma, v, M) \sim \mathcal{C}(\Sigma', v, M)}$$

$$\epsilon \sim \epsilon \qquad \frac{[\![F]\!] = \lfloor F' \rfloor \quad \Sigma \sim \Sigma' \quad \forall v,\ \models R\{r \leftarrow v\} : \texttt{analyze}(F)(l)}{\mathcal{F}(r, F.\texttt{code}, \sigma, l, R).\Sigma \sim \mathcal{F}(r, F'.\texttt{code}, \sigma, l, R).\Sigma'}$$

In the rule for stack frames, v stands for the value that will eventually be returned to the pending function call. Since it is unpredictable, we quantify universally over this return value. Semantic preservation for constant propagation follows by theorem 3 from the simulation lemma below.

Lemma 7 *If $G \vdash S_1 \xrightarrow{t} S_2$ and $S_1 \sim S_1'$, there exists S_2' such that $G' \vdash S_1' \xrightarrow{t} S_2'$ and $S_1' \sim S_2'$,*

7.3 Common subexpression elimination

7.3.1 Static analysis

Common subexpression elimination is implemented via local value numbering performed over extended basic blocks. Value numbers x are identifiers representing runtime values abstractly. A forward dataflow analysis associates to each program point l, F a pair (ϕ, η) of a partial mapping ϕ from registers to value numbers and a set η of equations between value numbers of the form $x = op(\vec{x})$ or $x = \kappa, mode(\vec{x})$. In a sense that will be made semantically precise below, the first equation form means that the operation op applied to concrete values matching the value numbers $\vec{x}$ returns a concrete value that matches x; likewise, the second equation means that computing an address using the addressing mode $mode$ applied to values matching $\vec{x}$ and loading a quantity κ from this address in the current memory state returns a value matching x. In addition to ϕ and η, the analysis result at each point also contains a supply of fresh value numbers and a proof that value numbers appearing in ϕ and η are not fresh with respect to this supply. We omit these two additional components for simplicity. The transfer function $T_F(l, A)$, where $A = (\phi, \eta)$ is the analysis result "before" point l, is defined as follows. If the instruction at l is a move $\texttt{op}(\texttt{move}, r_s, r_d, l')$, we record the equality $r_d = r_s$ by returning $(\phi\{r_d \leftarrow \phi(r_s)\}, \eta)$. If the instruction at l is $\texttt{op}(op, \vec{r}, r, l')$, we determine the value numbers $\vec{x} = \phi(\vec{r})$, associating fresh value numbers to elements of $\vec{r}$ if needed. We then check whether η contains an equation of the form $x = op(\vec{x})$ for some x. If so, the computation performed by this `op` operation has already been performed earlier in the program and the transfer function returns $(\phi\{r \leftarrow x\}, \eta)$. If not, we allocate a fresh value number x to stand for the result of this operation and return $(\phi\{r \leftarrow x\}, \eta \cup \{x = op(\vec{x})\})$ as the result of the transfer function. The definition of the transfer function is similar in the case of `load` instructions, using memory equalities $x = \kappa, mode(\vec{x})$ instead of arithmetic equalities $x = op(\vec{x})$. For `store` instructions, in the absence of nonaliasing information we must assume that the store can invalidate any of the memory equations that currently hold. The transfer function therefore removes all such equations from η. For `call` instructions, we remove all equations, since eliminating common subexpressions across function calls is often detrimental to the quality of register allocation. Other instructions keep (ϕ, η) unchanged. We say that a value numbering (ϕ, η) is satisfiable in a register state R and a memory state M, and we write $R, M \models \phi, \eta$, if there exists a valuation V associating concrete values to value numbers such that

1. If $\phi(r) = \lfloor x \rfloor$, then $R(r) = V(x)$.

2. If η contains the equation $x = op(\vec{x})$, then $\texttt{eval\_op}(op, V(\vec{x})) = \lfloor V(x) \rfloor$.
3. If η contains the equation $x = \kappa, mode(\vec{x})$, then there exists b and δ such that $\texttt{eval\_addressing}(mode, V(\vec{x})) = \lfloor \texttt{ptr}(b, \delta) \rfloor$ and $\texttt{load}(\kappa, M, b, \delta) = \lfloor V(x) \rfloor$.

Using this notion of satisfaction, we order value numberings by entailment: $(\phi', \eta') \geq (\phi, \eta)$ if $R, M \models \phi, \eta \implies R, M \models \phi', \eta'$ for all R, M. Least upper bounds for this ordering are known to be difficult to compute efficiently [48,38]. We sidestep this issue by solving the dataflow inequation using the approximate solver based on propagation described in section 7.1 instead of Kildall's algorithm.

7.3.2 Code transformation

The actual elimination of common subexpressions is simple. Consider an instruction i at point l, and let (ϕ, η) be the results of the static analysis at point l. If i is $\texttt{op}(op, \vec{r}, r, l')$ or $\texttt{load}(\kappa, mode, \vec{r}, r, l')$, and there exists a register r' such that the equation $\phi(r') = op(\phi(\vec{r}))$ or respectively $\phi(r') = \kappa, mode(\phi(\vec{r}))$ is in η, we rewrite the instruction as a move $\texttt{op}(\texttt{move}, r, r', l')$ from r' to r. This eliminates the redundant computation, reusing instead the result of the previous equivalent computation, which is still available in register r'. In all other cases, the instruction is unchanged.

7.3.3 Semantic preservation

The correctness proof for common subexpression elimination closely follows the pattern of the proof for constant propagation (see section 7.2.3), using a lock-step simulation diagram. The only difference is the replacement of the hypothesis $\models R : A(l)$ in the invariant between states by the hypothesis $R, M \models A(l)$, meaning that at every program point the current register and memory states must satisfy the value numbering obtained by static analysis.

8 Register allocation

The next compilation pass performs register allocation by coloring of an interference graph.

8.1 The target language: LTL

The target language for register allocation is a variant of RTL called LTL (Location Transfer Language). Syntactically, the only difference between LTL and RTL is that the RTL pseudo-registers r appearing as arguments and results of instructions are replaced by locations ℓ. A location is either a hardware register or an abstract designation of a stack slot.

Locations:	$\ell ::= r_m \mid s$	
Machine registers:	$r_m ::= \texttt{R3} \mid \texttt{R4} \mid \ldots$	PowerPC integer registers
	$\mid \texttt{F1} \mid \texttt{F2} \mid \ldots$	PowerPC float registers

$$\frac{F.\texttt{code}(l) = \lfloor \texttt{call}(sig, \ell_f, \vec{\ell}, \ell, l') \rfloor \qquad L(\ell_f) = \texttt{ptr}(b, 0) \qquad \texttt{funct}(G, b) = \lfloor Fd \rfloor \qquad Fd.\texttt{sig} = sig}{G \vdash \mathcal{S}(\Sigma, F, \sigma, l, L, M) \xrightarrow{\epsilon} \mathcal{C}(\mathcal{F}(\ell, F, \sigma, l', \texttt{postcall}(L)).\Sigma, Fd, L(\vec{\ell}), M)}$$

Fig. 12 Semantics of LTL. The transitions not shown are similar to those of RTL.

Stack slots:	$s ::= \texttt{local}(\tau, \delta)$	local variables
	$\mid \texttt{incoming}(\tau, \delta)$	incoming parameters
	$\mid \texttt{outgoing}(\tau, \delta)$	outgoing arguments

In stack slots, τ is the intended type of the slot (`int` or `float`) and δ an integer representing a word offset in one of the three areas of the activation record (`local`, `incoming` and `outgoing`).

In LTL, stack slots are not yet mapped to actual memory locations. Their values, along with those of the machine registers, are recorded in a mapping $L : \texttt{loc} \to \texttt{val}$ similar to the mapping $R : \texttt{reg} \to \texttt{val}$ used in the RTL semantics and disjoint from the memory state M. However, the LTL semantics treats locations in a way that anticipates their behaviors once they are later mapped to actual memory locations and processor registers. In particular, we account for the possible overlap between distinct stack slots once they are mapped to actual memory areas. For instance, $\texttt{outgoing}(\texttt{float}, 0)$ overlaps with $\texttt{outgoing}(\texttt{int}, 0)$ and $\texttt{outgoing}(\texttt{int}, 1)$: assigning to one of these locations invalidates the values of the other two. This is reflected in the weak "good variable" property for location maps L:

$$(L\{\ell_1 \leftarrow v\})(\ell_2) = \begin{cases} v & \text{if } \ell_1 = \ell_2; \\ L(\ell_2) & \text{if } \ell_1 \text{ and } \ell_2 \text{ do not overlap;} \\ \texttt{undef} & \text{if } \ell_1 \text{ and } \ell_2 \text{ partially overlap.} \end{cases}$$

Contrast with the standard "good variable" property for register maps R:

$$(R\{r_1 \leftarrow v\})(r_2) = \begin{cases} v & \text{if } r_1 = r_2; \\ R(r_2) & \text{if } r_1 \neq r_2. \end{cases}$$

The dynamic semantics of LTL is illustrated in figure 12. Apart from the use of location maps L and overlap-aware update instead of register maps R and normal updates, the only significant difference with the semantics of RTL is the semantics of `call` instructions. In preparation for enforcing calling conventions as described later in section 11, processor registers that are temporary or caller-save according to the calling conventions are set to `undef` in the location state of the caller, using the function `postcall` defined as

$$\texttt{postcall}(L)(\ell) = \begin{cases} \texttt{undef} & \text{if } \ell \text{ is a temporary or caller-save register;} \\ L(\ell) & \text{otherwise.} \end{cases}$$

This forces the LTL producer, namely the register allocation pass, to ensure that no value live across a function call is stored in a caller-save register.

8.2 Code transformation

For every function, register allocation is performed in four steps, which we now outline.

8.2.1 Type reconstruction for RTL

The first step performs type reconstruction for the RTL source function in a trivial "`int`-or-`float`" type system similar to that of Cminor (see section 4.3). To each RTL pseudo-register we assign a type `int` or `float`, based on its uses within the function. The resulting type assignment $\Gamma : r \mapsto \tau$ will guide the register allocator, making sure a machine register or stack location of the correct kind is assigned to each pseudo-register.

All operators, addressing modes and conditions are monomorphic (each of their arguments has only one possible type, either `int` or `float`), except the `move` operation which is polymorphic with type $\forall\tau.\ \tau \to \tau$. Type reconstruction can therefore be performed by trivial unification. We use the verified validator approach: a candidate type assignment Γ is computed by untrusted Caml code, using in-place unification, then verified for correctness by a simple type-checker, written and proved correct in Coq.

8.2.2 Liveness analysis

We compute the set $A(l)$ of pseudo-registers live "after" every program point l. These sets are a solution of the following backward dataflow inequations:

$$A(l) \supseteq T(s, A(s)) \text{ for all } s \text{ successor of } l$$

The transfer function T computes the set of live pseudo-registers "before" an instruction, as a function of this instruction and the set of live pseudo-registers "after" it. Classically, it removes the registers defined by the instruction, then adds the registers used. A special case is made for `op` or `load` instructions whose result register is not live "after" the instruction, since these instructions will later be eliminated as dead code. For instance, if the instruction at l in F is $\texttt{op}(op, \vec{r}, r, l')$, then

$$T(l, A) = \begin{cases} (A \setminus \{r\}) \cup \vec{r} & \text{if } r \in A; \\ A & \text{if } r \notin A. \end{cases}$$

Liveness information is computed by Kildall's algorithm, using the generic dataflow solver described in section 7.1.

8.2.3 Construction of the interference graph

Based on the results of liveness analysis, an interference graph is built following Chaitin's construction [20]. Two kinds of interferences are recorded: between two pseudo-registers (r, r') and between a pseudo-register r and a machine register r_m. To enable coalescing during allocation, register affinities arising from moves (either explicit or implicit through calling conventions) are also recorded. (Affinities do not affect the correctness of the generated code but have a considerable impact on its performance.)

The interference graph is represented by sets of edges: either unordered pairs (r, r') or ordered pairs (r, r_m). The graph is constructed incrementally by enumerating every RTL instruction and adding interference edges between the defined register r (if any) and the registers $r' \in A(l) \setminus \{r\}$ live "across" the instruction. For move instructions $l : \texttt{op}(\texttt{move}, r_s, r_d, l')$, we avoid adding an edge between r_s and r_d, as proposed by Chaitin [20]. Finally, for call instructions, additional interference edges are introduced pairwise between the registers live across the call and the caller-save machine registers.

8.2.4 Coloring of the interference graph

To color the interference graph, we use the iterated coalescing algorithm of George and Appel [35]. The result is a mapping $\Phi : r \mapsto \ell$ from pseudo-registers to locations. Following once more the verified validator approach, the actual coloring is performed by an untrusted implementation of the George-Appel algorithm written in Caml and using imperative doubly-linked lists for efficiency. The candidate coloring returned is then verified by a simple validator written and proved correct in Coq. Like many NP-complete problems, graph coloring is a paradigmatic example of an algorithm that is significantly easier to validate a posteriori than to prove correct. Validation proceeds by enumerating the nodes and edges of the interference graph, checking the following properties:

1. Correct colors: $\Phi(r) \neq \Phi(r')$ for all edges (r, r') of the interference graph; likewise, $\Phi(r) \neq r_m$ for all interference edges (r, m).
2. Register class preservation: the type of the location $\Phi(r)$ is equal to $\Gamma(r)$ for all pseudo-registers r.
3. Validity of locations: for all r, the location $\Phi(r)$ is either a `local` stack slot or a non-temporary machine register[5].

8.2.5 LTL generation

Finally, the actual code transformation from RTL to LTL is a trivial per-instruction rewriting of the CFG where each mention of a pseudo-register r is replaced by the location $\Phi(r)$ allocated to r. For instance, the RTL instruction $l : \texttt{op}(op, \vec{r}, r, l')$ becomes the LTL instruction $l : \texttt{op}(op, \Phi(\vec{r}), \Phi(r), l')$. There are two exceptions to this rule. First, a move instruction $l : \texttt{op}(\texttt{move}, r_s, r_d, l')$ such that $\Phi(r_s) = \Phi(r_d)$ is turned into a no-operation $l : \texttt{nop}(l')$, therefore performing one step of coalescing. Second, an `op` or `load` instruction whose result register is not live after the instruction is similarly turned into a `nop` instruction, therefore performing dead-code elimination.

8.3 Semantic preservation

The proof that register allocation preserves program behaviors is, once more, based on a lock-step simulation diagram. The invariant between states is, however, more complex than those used for constant propagation or common subexpression elimination. For these two optimizations, the register state R was identical between matching states, because the same values would be computed (by possibly different instructions) in the original and transformed program and would be stored in the same registers. This assumption no longer holds in the case of register allocation.

A value computed by the original RTL program and stored in pseudo-register r is stored in location $\Phi(r)$ in the transformed LTL program. Naively, we could relate the RTL register state R and the LTL location state L by $R(r) = L(\Phi(r))$ for all pseudo-registers r. However, this requirement is too strong, as it essentially precludes sharing a location between several pseudo-registers.

[5] We reserve 2 integer and 3 float machine registers as temporaries to be used later for spilling and reloading. These temporary registers must therefore not be used by the register allocator.

To progress towards the correct invariant, consider the semantic interpretation of live and dead pseudo-registers. If a pseudo-register r is dead at point l in the original RTL code, then its value $R(r)$ has no impact on the remainder of the program execution: either r will never be used again, or it will be redefined before being used; in either case, its value $R(r)$ could be replaced by any other value without any harm. A better relation between the values of pseudo-registers R and locations L at point l is therefore

$$R(r) = L(\Phi(r)) \text{ for all pseudo-registers } r \in T(l, A(l))$$

In other words, at each program point l, register allocation must preserve the values of all registers live "before" executing the instruction at this point. This property that we have never seen spelled out explicitly in compiler literature captures concisely and precisely the essence of register allocation.

The invariant between RTL and LTL states is, then:

$$\frac{\begin{array}{c}\Sigma \sim \Sigma' \quad [\![F]\!] = \lfloor F' \rfloor \quad R(r) = L(\Phi(r)) \text{ for all } r \in T(l, A(l)) \\ \texttt{typecheck}(F) = \lfloor \Gamma \rfloor \quad \texttt{analyze}(F) = \lfloor A \rfloor \quad \texttt{regalloc}(F, A, \Gamma) = \lfloor \Phi \rfloor\end{array}}{\mathcal{S}(\Sigma, F.\texttt{code}, \sigma, l, R, M) \sim \mathcal{S}(\Sigma', F', \sigma, l, L, M)}$$

$$\frac{\Sigma \sim \Sigma' \quad [\![Fd]\!] = \lfloor Fd' \rfloor}{\mathcal{C}(\Sigma, Fd, \vec{v}, M) \sim \mathcal{C}(\Sigma', Fd', \vec{v}, M)} \qquad \frac{\Sigma \sim \Sigma'}{\mathcal{R}(\Sigma, v, M) \sim \mathcal{R}(\Sigma', v, M)}$$

The invariant relating frames in the call stacks is similar to that relating regular states, with a universal quantification on the return value, as in section 7.2.3. The proof of the lock-step simulation diagram makes heavy use of the definition of the transfer function T for liveness analysis, combined with the following characterization of the register allocation Φ with respect to the results A of liveness analysis:

- For a move instruction $l : \texttt{op}(\texttt{move}, r_s, r_d, l')$, we have $\Phi(r_d) \neq \Phi(r')$ for all $r' \in A(l) \setminus \{r_s, r_d\}$.
- For other `op`, `load` or `call` instructions at l with destination register r, we have $\Phi(r) \neq \Phi(r')$ for all $r' \in A(l) \setminus \{r\}$.
- For call instructions $l : \texttt{call}(sig, (r_f \mid id), \vec{r}, r_d, l')$, we additionally have $\Phi(r') \neq r_m$ for all $r' \in A(l) \setminus \{r_d\}$ and all callee-save registers r_m.

9 Branch tunneling and no-op elimination

Register coalescing and dead-code elimination, performed in the course of register allocation, generate a number of `nop` instructions. Now is a good time to "short-circuit" them, rendering them unreachable and making them candidates for removal during CFG linearization (section 10). Since `nop` in a CFG representation also encodes unconditional branches, this transformation also performs branch tunneling: the elimination of branches to branches.

9.1 Code transformation

Branch tunneling rewrites the LTL control-flow graph, replacing each successor, i.e. the l' and l'' in instructions such as $\texttt{op}(op, \vec{\ell}, \ell, l')$ or $\texttt{cond}(cond, \vec{\ell}, l', l'')$, by its effective

destination $D_F(l')$. Naively, the effective destination is computed by chasing down sequences of `nop` instructions, stopping at the first non-`nop` instruction:

$$D_F(l) = \begin{cases} D_F(l') & \text{if } F.\texttt{code}(l) = \lfloor \texttt{nop}(l') \rfloor; \\ l & \text{otherwise.} \end{cases}$$

This is not a proper definition: if the control-flow graph contains cycles consisting only of `nop` instructions, such as $l : \texttt{nop}(l)$, the computation of $D_F(l)$ fails to terminate.

A simple solution is to bound the recursion depth when computing $D_F(l)$, returning l when the counter reaches 0. The initial value N can be chosen at will; the number of instructions in the code of function F is a good choice.

A more elegant solution, suggested by an anonymous reviewer, uses a union-find data structure U_F. A first scan of the control-flow graph populates U_F by adding an edge from l to l' for each instruction $l : \texttt{nop}(l')$, provided l and l' are not already in the same equivalence class. The effective destination $D_F(l)$, then, is defined as the canonical representative of l in the union-find structure U_F.

9.2 Semantic preservation

Semantic preservation for branch tunneling follows from a simulation diagram of the "option" kind (see figure 4). Intuitively, the execution of a non-call, non-return instruction that causes a transition from point l_1 to point l_2 in the original code corresponds to the execution of zero or one instructions in the tunneled code, from point $D_F(l_1)$ to point $D_F(l_2)$. The "zero" case can appear for example when the instruction at l_1 is $\texttt{nop}(l_2)$. The definition of the invariant between execution states is therefore

$$\frac{[\![F]\!] = \lfloor F' \rfloor \qquad \Sigma \sim \Sigma'}{\mathcal{S}(\Sigma, F, \sigma, l, L, M) \sim \mathcal{S}(\Sigma, F', \sigma, D_F(l), L, M)}$$

for regular states, with the obvious definitions for call states, return states, and stack frames. For simulation diagrams of the "option" kind, we must provide a measure to show that it is not possible to take infinitely many "or zero" cases. For branch tunneling, a suitable measure is the number of `nop` instructions that are skipped starting with the current program point:

$$|\mathcal{S}(\Sigma, F, sp, l, L, M)| = \#\texttt{nop}_F(l)$$

$$\#\texttt{nop}_F(l) = \begin{cases} 1 + \#\texttt{nop}_F(l') & \text{if } F.\texttt{code}(l) = \lfloor \texttt{nop}(l') \rfloor \text{ and } D_F(l) \neq l; \\ 0 & \text{otherwise.} \end{cases}$$

This definition is well founded because of the way D_F is constructed from the union-find structure U_F. The crucial property of the D and $\#\texttt{nop}$ functions is that, for all program points l,

$$D_F(l) = l \ \vee \ \exists l', \ F.\texttt{code}(l) = \lfloor \texttt{nop}(l') \rfloor \wedge D_F(l') = D_F(l) \wedge \#\texttt{nop}_F(l') < \#\texttt{nop}_F(l)$$

Theorem 6 *If $S_1 \sim S_2$ and $G \vdash S_1 \xrightarrow{t} S_1'$, either there exists S_2' such that $G' \vdash S_2 \xrightarrow{t} S_2'$ and $S_1' \sim S_2'$, or $|S_1'| < |S_1|$ and $t = \epsilon$ and $S_1' \sim S_2$.*

10 Linearization of the control-flow graph

The next compilation step linearizes the control-flow graphs of LTL, replacing them by lists of instructions with explicit labels and unconditional and conditional branches to labels, in the style of assembly code. While the CFG representation of control is very convenient for performing dataflow analyses, the linearized representation makes it easier to insert new instructions, as needed by some of the subsequent passes.

Discussions of linearization in the literature focus on trace picking heuristics that reduce the number of jumps introduced, but consider the actual production of linearized code trivial. Our first attempts at proving directly the correctness of a trace picking algorithm that builds linearized code and performs branch tunneling on the fly showed that this is not so trivial. We therefore perform tunneling in a separate pass (see section 9), then implement CFG linearization in a way that clearly separates heuristics from the actual production of linearized code.

10.1 The target language: LTLin

The target language for CFG linearization is LTLin, a variant of LTL where control-flow graphs are replaced by lists of instructions.

LTLin instructions:

i	$::= \texttt{op}(op, \vec{\ell}, \ell)$	arithmetic operation
	$\mid \texttt{load}(\kappa, mode, \vec{\ell}, \ell)$	memory load
	$\mid \texttt{store}(\kappa, mode, \vec{\ell}, \ell)$	memory store
	$\mid \texttt{call}(sig, (\ell \mid id), \vec{\ell}, \ell)$	function call
	$\mid \texttt{tailcall}(sig, (\ell \mid id), \vec{\ell})$	function tail call
	$\mid \texttt{cond}(cond, \vec{\ell}, l_{true})$	conditional branch
	$\mid \texttt{goto}(l)$	unconditional branch
	$\mid \texttt{label}(l)$	definition of the label l
	$\mid \texttt{return} \mid \texttt{return}(\ell)$	function return

LTLin code sequences:

c	$::= i_1 \dots i_n$	list of instructions

LTLin functions:

F	$::= \{\ \texttt{sig} = sig;$	
	$\quad \texttt{params} = \vec{\ell};$	parameters
	$\quad \texttt{stacksize} = n;$	size of stack data block
	$\quad \texttt{code} = c\}$	instructions

The dynamic semantics of LTLin is similar to that of LTL as far as the handling of data is concerned. In execution states, program points within a function F are no longer represented by CFG labels l, but instead are represented by code sequences c that are suffixes of $F.\texttt{code}$. The first element of c is the instruction to execute next. As shown in figure 13, most instructions have "fall-through" behavior: they transition from $i.c$ to c. To resolve branches to a label l, we use the auxiliary function $\texttt{findlabel}(F, l)$ that returns the maximal suffix of $F.\texttt{code}$ that starts with $\texttt{label}(l)$, if it exists.

$$\frac{\texttt{eval\_op}(G, \sigma, op, L(\vec{\ell})) = \lfloor v \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{op}(op, \vec{\ell}, \ell).c, L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, L\{\ell \leftarrow v\}, M)}$$

$$\frac{\texttt{findlabel}(F, l_{true}) = \lfloor c' \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{goto}(l).c, L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c', L, M)}$$

$$\frac{\texttt{eval\_cond}(cond, L(\vec{\ell})) = \lfloor \texttt{true} \rfloor \quad \texttt{findlabel}(F, l_{true}) = \lfloor c' \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{cond}(cond, \vec{\ell}, l_{true}).c, L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c', L, M)}$$

$$\frac{\texttt{eval\_cond}(cond, L(\vec{\ell})) = \lfloor \texttt{false} \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{cond}(cond, \vec{\ell}, l_{true}).c, L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, L, M)}$$

Fig. 13 Semantics of LTLin (selected rules).

10.2 Code transformation

CFG linearization is performed in two steps that clearly separate the heuristic, correctness-irrelevant part of linearization from the actual, correctness-critical code generation part [1, chap. 8]. We first compute an enumeration $l_1 \ldots l_n$ of the labels of the CFG nodes reachable from the entry node. The order of labels in this enumeration dictates the positions of the corresponding instructions in the list of LTLin instructions that we will generate. Following the verified validator approach, this enumeration of CFG nodes is computed by untrusted code written in Caml. It can implement any of the textbook heuristics for picking "hot" traces that should execute without branches, including static branch prediction. This enumeration is validated by a validator written in Coq that checks the following two conditions:

1. No node l appears twice in the enumeration.
2. All nodes l reachable from the function entry point appear in the enumeration.

For condition 2, the validator precomputes the set of reachable nodes using a trivial forward dataflow analysis, where the abstract domain is $\{\texttt{false}, \texttt{true}\}$ and the transfer function is $T(l, a) = a$. (By definition, all successors of a reachable node are reachable).

To generate LTLin code, we then concatenate the instructions of the control-flow graph in the order given by the enumeration $l_1, \ldots, l_n$. Each instruction is preceded by $\texttt{label}(l_i)$ and followed by a `goto` to its successor, unless this `goto` is unnecessary because it would branch to an immediately following `label`. Formally, the basic code generation function is of the form $C(i, c)$ and returns a sequence c' of instructions obtained by prepending the translation of the LTL instruction i to the initial instruction sequence c. Here are some representative cases:

$$\begin{aligned}
C(\texttt{op}(op, \vec{\ell}, \ell, l'), c) &= \texttt{op}(op, \vec{\ell}, \ell).c \quad \text{if } c \text{ starts with } \texttt{label}(l');\\
C(\texttt{op}(op, \vec{\ell}, \ell, l'), c) &= \texttt{op}(op, \vec{\ell}, \ell).\texttt{goto}(l').c \quad \text{otherwise.}\\
C(\texttt{cond}(cond, \vec{\ell}, l_{true}, l_{false}), c) &= \texttt{cond}(cond, \vec{\ell}, l_{true}).c\\
&\quad \text{if } c \text{ starts with } \texttt{label}(l_{false});\\
C(\texttt{cond}(cond, \vec{\ell}, l_{true}, l_{false}), c) &= \texttt{cond}(\neg cond, \vec{\ell}, l_{false}).c\\
&\quad \text{if } c \text{ starts with } \texttt{label}(l_{true});\\
C(\texttt{cond}(cond, \vec{\ell}, l_{true}, l_{false}), c) &= \texttt{cond}(cond, \vec{\ell}, l_{true}).\texttt{goto}(l_{false}).c\\
&\quad \text{otherwise.}
\end{aligned}$$

This function is then iterated over the enumeration $\vec{l}$ of CFG nodes, inserting the appropriate `label` instructions:

$$C(F, \epsilon) = \epsilon$$
$$C(F, l.\vec{l}) = \texttt{label}(l).C(i, C(F, \vec{l})) \text{ if } F.\texttt{code}(l) = \lfloor i \rfloor$$

10.3 Semantic preservation

Each intra-function transition in the original LTL code corresponds to 2 or 3 transitions in the generated LTLin code: one to skip the `label` instruction, one to execute the actual instruction, and possibly one to perform the `goto` to the successor. The proof of semantic preservation is therefore based on a simulation diagram of the "plus" kind. The main invariant is: whenever the LTL program is at program point l in function F, the LTLin program is at the instruction sequence $\texttt{findlabel}(F', l)$ in the translation F' of F.

$$\frac{[\![F]\!] = \lfloor F' \rfloor \qquad \Sigma \sim \Sigma' \qquad \texttt{findlabel}(F', l) = \lfloor c \rfloor \qquad l \text{ is reachable from } F.\texttt{entrypoint}}{\mathcal{S}(\Sigma, F, \sigma, l, L, M) \sim \mathcal{S}(\Sigma', F', \sigma, c, L, M)}$$

A pleasant surprise is that the simulation proof goes through under very weak assumptions about the enumeration of CFG nodes produced by the external heuristics: conditions (1) and (2) of section 10.2 are all it takes to guarantee semantic preservation. This shows that our presentation of linearization is robust: many trace picking heuristics can be tried without having to redo any of the semantic preservation proofs.

11 Spilling, reloading, and materialization of calling conventions

The next compilation pass finishes the register allocation process described in section 8 by inserting explicit "spill" and "reload" operations around uses of pseudo-registers that have been allocated stack slots. Additionally, calling conventions are materialized in the generated code by inserting moves to and from the conventional locations used for parameter passing around function calls.

11.1 The target language: Linear

Linear, the target language for this pass, is a variant of LTLin where the operands of arithmetic operations, memory accesses and conditional branches are restricted to machine registers instead of arbitrary locations. This is consistent with the RISC instruction set of our target processor. (Machine registers, written r_m so far, will now be written r for simplicity.) Two instructions `getstack` and `setstack` are provided to move data between machine registers and stack slots s.

Linear instructions:

i	::= $\texttt{getstack}(s, r) \mid \texttt{setstack}(r, s)$	reading, writing a stack slot
	$\mid \texttt{op}(op, \vec{r}, r)$	arithmetic operation
	$\mid \texttt{load}(\kappa, mode, \vec{r}, r)$	memory load
	$\mid \texttt{store}(\kappa, mode, \vec{r}, r)$	memory store
	$\mid \texttt{call}(sig, (r \mid id))$	function call
	$\mid \texttt{tailcall}(sig, (r \mid id))$	function tail call
	$\mid \texttt{cond}(cond, \vec{r}, l_{true})$	conditional branch
	$\mid \texttt{goto}(l)$	unconditional branch
	$\mid \texttt{label}(l)$	definition of the label l
	$\mid \texttt{return}$	function return

Linear code sequences:

c	::= $i_1 \ldots i_n$	list of instructions

Linear functions:

F	::= { $\texttt{sig} = sig$;	
	$\texttt{stacksize} = n$;	size of stack data block
	$\texttt{code} = c$}	instructions

Another novelty of Linear is that `call`, `tailcall` and `return` instructions, as well as function definitions, no longer carry a list of locations for their parameters or results: the generated Linear code contains all the necessary move instructions to ensure that these parameters and results reside in the locations determined by the calling conventions. Correspondingly, call states and return states no longer carry lists of values: instead, they carry full location maps L where the values of arguments and results can be found at conventional locations, determined as a function of the signature of the called function.

Program states:	S	::= $\mathcal{S}(\Sigma, F, \sigma, c, L, M)$	regular state
		$\mid \mathcal{C}(\Sigma, Fd, L, M)$	call state
		$\mid \mathcal{R}(\Sigma, L, M)$	return state
Call stacks:	Σ	::= $(\mathcal{F}(F, \sigma, c, L))^*$	list of frames

As shown in the dynamic semantics for Linear (see figure 14), the behavior of locations across function calls is specified by two functions: $\texttt{entryfun}(L)$ determines the locations on entrance to the callee as a function of the locations L before the `call`, and $\texttt{exitfun}(L, L')$ determines the locations in the callee when the `call` returns as a function of the caller's locations before the call, L, and the callee's locations before the return, L'. In summary, processor registers are global, but some are preserved by the callee; `local` and `incoming` slots of the caller are preserved across the call; and the `incoming` slots on entrance to the callee are the `outgoing` slots of the caller.

Location l	$\texttt{entryfun}(L)(l)$	$\texttt{exitfun}(L, L')(l)$
r	$L(r)$	$L(r)$ if r is callee-save
		$L'(r)$ if r is caller-save
$\texttt{local}(\tau, \delta)$	`undef`	$L(\texttt{local}(\tau, \delta))$
$\texttt{incoming}(\tau, \delta)$	$L(\texttt{outgoing}(\tau, \delta))$	$L(\texttt{incoming}(\tau, \delta))$
$\texttt{outgoing}(\tau, \delta)$	`undef`	$L'(\texttt{incoming}(\tau, \delta))$

$$\frac{\texttt{eval\_op}(G, \sigma, op, L(\vec{r})) = \lfloor v \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{op}(op, \vec{r}, r).c, L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, L\{r \leftarrow v\}, M)}$$

$$\frac{L(r) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{call}(sig, r).c, L, M) \xrightarrow{\epsilon} \mathcal{C}(\mathcal{F}(F, \sigma, c, L).\Sigma, Fd, L, M)}$$

$$\frac{L(r) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig \quad L' = \texttt{exitfun}(\Sigma.\texttt{top}.L, L)}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{tailcall}(sig, r).c, L, M) \xrightarrow{\epsilon} \mathcal{C}(\Sigma, Fd, L', M)}$$

$$\frac{L' = \texttt{exitfun}(\Sigma.\texttt{top}.L, L)}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{return}.c, L, M) \xrightarrow{\epsilon} \mathcal{R}(\Sigma, L', \texttt{free}(M, \sigma))}$$

$$\frac{\texttt{alloc}(M, 0, F.\texttt{stacksize}) = (\sigma, M') \quad L' = \texttt{entryfun}(L)}{G \vdash \mathcal{C}(\Sigma, \texttt{internal}(F), L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F.\texttt{code}, \sigma, F.\texttt{code}, L', M')}$$

$$\frac{\vdash Fe(\vec{v}) \stackrel{t}{\Rightarrow} v \quad \vec{v} = L(\texttt{loc\_arguments}(Fe.\texttt{sig})) \quad L' = L\{\texttt{loc\_result}(Fe.\texttt{sig}) \leftarrow v\}}{G \vdash \mathcal{C}(\Sigma, \texttt{external}(Fe), L, M) \xrightarrow{t} \mathcal{R}(\Sigma, L', M)}$$

$$G \vdash \mathcal{R}(\mathcal{F}(F, \sigma, c, L_0).\Sigma, L, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, L, M)$$

Fig. 14 Semantics of Linear. The transitions not shown are similar to those of LTLin.

In other words, the `entryfun` and `exitfun` anticipate, at the level of the Linear semantics, the effect of future transformations (placement of stack slots in memory and insertion of function prologues and epilogues to save and restore callee-save registers) performed in the next compilation pass (section 12).

In the rules for `tailcall` and `return` of figure 14, the notation $\Sigma.\texttt{top}.L$ stands for the L component of the top frame in stack Σ. A suitable default is defined for an empty stack.

$$(\mathcal{F}(F, \sigma, c, L).\Sigma).\texttt{top}.L = L \qquad \epsilon.\texttt{top}.L = (\ell \mapsto \texttt{undef})$$

11.2 Code transformation

Our strategy for spilling and reloading is simplistic: each use of a spilled pseudo-register is preceded by a `getstack` instruction to reload the pseudo-register in a machine register, and each definition is followed by a `setstack` instruction that spills the result. No attempt is made to reuse a reloaded value, nor to delay spilling. We reserve 3 integer registers and 3 float registers to hold reloaded values and results of instructions before spilling. (This does not follow compiler textbooks, which prescribe re-running register allocation to assign registers to reload and spill temporaries. However, it is difficult to prove termination for this practice, and moreover it requires semantic reasoning about partially-allocated code.) The following case should give the flavor of the transformation:

$$\begin{aligned}[\![\texttt{op}(op, \vec{\ell}, \ell).c]\!] = \texttt{let } \vec{r} = \texttt{regs\_for}(\vec{\ell}) \texttt{ and } r = \texttt{reg\_for}(\ell) \texttt{ in} \\ \texttt{reloads}(\vec{\ell}, \vec{r}).\texttt{op}(op, \vec{r}, r).\texttt{spill}(r, \ell).[\![c]\!]\end{aligned}$$

$\texttt{reg\_for}(\ell)$ returns r if the location ℓ is a machine register r, or a temporary register of the appropriate type if ℓ is a stack slot. $\texttt{regs\_for}(\vec{\ell})$ does the same for a list of locations, using different temporary registers for each location. $\texttt{spill}(r, \ell)$ generates the `move` or

`setstack` operation that sets ℓ to the value of register r. Symmetrically, $\texttt{reloads}(\vec{\ell}, \vec{r})$ generates the `move` or `getstack` operations that set $\vec{r}$ to the values of locations $\vec{\ell}$.

For `call` and `tailcall` instructions with signature sig and arguments $\vec{\ell}$, we insert moves from $\vec{\ell}$ to the locations dictated by the calling conventions. These locations (a mixture of processor register and `outgoing` stack slots) are determined as a function $\texttt{loc\_arguments}(sig)$ of the signature sig of the called function. Likewise, the result of the call, which is passed back in the conventional location $\texttt{loc\_result}(sig)$, is moved to the result location of the `call`.

$$\begin{aligned}[\![\texttt{call}(sig, id, \vec{\ell}, \ell).c]\!] = \texttt{parallel\_move}(\vec{\ell}, \texttt{loc\_arguments}(sig)).\\ \texttt{call}(sig, id).\texttt{spill}(\texttt{loc\_result}(sig), \ell).[\![c]\!]\end{aligned}$$

Symmetrically, at entry to a function F, we insert moves from $\texttt{loc\_parameters}(F.\texttt{sig})$ to $F.\texttt{params}$. ($\texttt{loc\_parameters}(sig)$ is $\texttt{loc\_arguments}(sig)$ where `outgoing` slots are replaced by `incoming` slots.)

The moves inserted for function arguments and for function parameters must implement a parallel assignment semantics: some registers can appear both as sources and destinations, as in $(r_1, r_2, r_3) := (r_2, r_1, r_4)$. It is folklore that such parallel moves can be implemented by a sequence of elementary moves using at most one temporary register of each kind. Formulating the parallel move algorithm in Coq and proving its correctness was a particularly difficult part of this development. The proof is detailed in a separate paper [85].

11.3 Semantic preservation

The correctness proof for the spilling pass is surprisingly involved because it needs to account for the fact that the location states L and memory states M differ significantly between the original LTLin code and the generated Linear code. An inessential source of difference is that the Linear code makes use of temporary registers and `outgoing` and `incoming` stack locations while the LTLin code does not. A deeper difference comes from the fact that, in LTLin, locations other than function parameters are consistently initialized to the `undef` value, while in Linear some of these locations (e.g. processor registers) just keep whatever values they had in the caller before the `call` instruction. Performing arithmetic over this `undef` value would cause the original LTLin program to go wrong, but it can still pass this value around, store it in memory locations, and read it back. Therefore, some `undef` values found in LTLin register and memory states can become any other value in the corresponding Linear location and memory states. To capture this fact, we use the "less defined than" ordering $\leq$ between values defined by

$$v \leq v' \quad \stackrel{\text{def}}{=} \quad v = v' \text{ or } v = \texttt{undef}$$

and extended to memory states as follows:

$$\begin{aligned}M \leq M' \quad \stackrel{\text{def}}{=} \quad \forall \kappa, b, \delta, v, \texttt{load}(\kappa, M, b, \delta) = \lfloor v \rfloor\\ \Rightarrow \exists v', \texttt{load}(\kappa, M', b, \delta) = \lfloor v' \rfloor \wedge v \leq v'\end{aligned}$$

Leroy and Blazy [59, section 5.3] study the properties of this relation between memory states. It commutes nicely with the `store`, `alloc` and `free` operations over memory states.

Putting it all together, we define agreement $L \leq L'$ between a LTLin location state L and a Linear location state L' as

$$L \leq L' \quad \stackrel{\text{def}}{=} \quad L(\ell) \leq L'(\ell) \text{ for all non-temporary registers or } \texttt{local} \text{ stack slots } \ell$$

and define the invariant relating LTLin and Linear execution states as follows.

$$\frac{\Sigma \sim \Sigma' : F.\texttt{sig} \qquad L \leq L' \qquad M \leq M'}{\mathcal{S}(\Sigma, F, \sigma, c, L, M) \sim \mathcal{S}(\Sigma', [\![F]\!], \sigma, [\![c]\!], L', M')}$$

$$\frac{\Sigma \sim \Sigma' : Fd.\texttt{sig} \qquad \vec{v} \leq L'(\texttt{loc\_arguments}(Fd.\texttt{sig})) \qquad L' \approx \Sigma.\texttt{top}.L \qquad M \leq M'}{\mathcal{C}(\Sigma, Fd, \vec{v}, M) \sim \mathcal{C}(\Sigma', Fd, L', M')}$$

$$\frac{\Sigma \sim \Sigma' : sig \qquad v \leq L'(\texttt{loc\_result}(sig)) \qquad L' \approx \Sigma.\texttt{top}.L \qquad M \leq M'}{\mathcal{R}(\Sigma, v, M) \sim \mathcal{R}(\Sigma', L', M')}$$

For call states and return states, the second premises capture the fact that the argument and return values can indeed be found in the corresponding locations dictated by the calling conventions, modulo the $\leq$ relation between values. The third premises $L' \approx \Sigma.\texttt{top}.L$ say that the current location state L' and that of the caller $\Sigma.\texttt{top}.L$ assign the same values to any callee-save location. Finally, in the invariant $\Sigma \sim \Sigma' : sig$ relating call stacks, a call signature sig is threaded through the call stack to make sure that the caller and the callee agree on the result type of the call, and therefore on the location used to pass the return value.

$$\frac{sig.\texttt{res} = \lfloor \texttt{int} \rfloor}{\epsilon \sim \epsilon : sig}$$

$$\frac{\Sigma \sim \Sigma' : F.\texttt{sig} \qquad c' = \texttt{spill}(\texttt{loc\_result}(sig), \ell).[\![c]\!] \qquad \texttt{postcall}(L) \leq L'}{\mathcal{F}(\ell, F, \sigma, \texttt{postcall}(L), c).\Sigma \sim \mathcal{F}([\![F]\!], \sigma, L', c').\Sigma' : sig}$$

Armed with these definitions and the proof of the parallel move algorithm of [85], we prove semantic preservation for this pass using a simulation diagram of the "star" kind. The only part of the transformation that could cause stuttering is the elimination of a redundant move from ℓ to ℓ. To prove that stuttering cannot happen, it suffices to note that the length of the LTLin instruction sequence currently executing decreases in this case.

12 Construction of the activation record

The penultimate compilation pass lays out the activation record for each function, allocating space for stack slots and turning accesses to slots into memory loads and stores. Function prologues and epilogues are added to preserve the values of callee-save registers.

12.1 The target language: Mach

The last intermediate language in our gentle descent towards assembly language is called Mach. It is a variant of Linear where the three infinite supplies of stack slots (local, incoming and outgoing) are mapped to actual memory locations in the stack frames of the current function (for local and outgoing slots) or the calling function (for incoming slots).

Mach instructions:	$i ::=$	$\texttt{setstack}(r, \tau, \delta)$	register to stack move
	$\mid$	$\texttt{getstack}(\tau, \delta, r)$	stack to register move
	$\mid$	$\texttt{getparent}(\tau, \delta, r)$	caller's stack to register move
	$\mid$	$\ldots$	as in Linear

In the three new move instructions, τ is the type of the data moved and δ its word offset in the corresponding activation record.

Mach code:	$c ::=$	$i_1 \ldots i_n$	list of instructions
Mach functions:	$F ::= \{$	$\texttt{sig} = sig;$	
		$\texttt{stack\_high} = n;$	upper bound of stack data block
		$\texttt{stack\_low} = n;$	lower bound of stack data block
		$\texttt{retaddr} = \delta;$	offset of saved return address
		$\texttt{link} = \delta;$	offset of back link
		$\texttt{code} = c\}$	instructions

Functions carry two byte offsets, `retaddr` and `link`, indicating where in the activation record the function prologue should save the return address into its caller and the back link to the activation record of its caller, respectively.

Program states:	$S ::=$	$\mathcal{S}(\Sigma, F, \sigma, c, R, M)$	regular states
	$\mid$	$\mathcal{C}(\Sigma, Fd, R, M)$	call states
	$\mid$	$\mathcal{R}(\Sigma, R, M)$	return states
Call stacks:	$\Sigma ::=$	$(\mathcal{F}(F, \sigma, ra, c))^*$	list of frames

Semantically, the main difference between Linear and Mach is that, in Mach, the register state R mapping processor registers to values is global and shared between caller and callee. In particular, R is not saved in the call stack, and as shown in figure 15, there is no automatic restoration of callee-save registers at function return; instead, the Mach code generator produces appropriate `setstack` and `getstack` instructions to save and restore used callee-save registers at function prologues and epilogues.

The `setstack` and `getstack` instructions are interpreted as memory stores and loads relative to the stack pointer. We write $\kappa(\tau)$ for the memory quantity appropriate for storing a value of type τ without losing information, namely $\kappa(\texttt{int}) = \texttt{int32}$ and $\kappa(\texttt{float}) = \texttt{float64}$. For the `getparent` instruction, we recover a pointer to the caller's stack frame by loading from the `link` location of our stack frame, then load from this pointer. This additional indirection is needed since, in our memory model, the callee's stack frame is not necessarily adjacent to that of the caller. This linking of stack frames is implemented by the rule for function entry.

The rules for `call` and function entry also make provisions for saving a return address within the caller's function code in the `retaddr` location of the activation record. This return address (a pointer to an instruction within a code block) becomes

$$\texttt{loadstack}(\tau, M, \sigma, \delta) = \begin{cases} \texttt{load}(\kappa(\tau), M, b, \delta' + \delta) & \text{if } \sigma = \texttt{ptr}(b, \delta') \\ \emptyset & \text{otherwise} \end{cases}$$

$$\texttt{storestack}(\tau, M, \sigma, \delta, v) = \begin{cases} \texttt{store}(\kappa(\tau), M, b, \delta' + \delta, v) & \text{if } \sigma = \texttt{ptr}(b, \delta') \\ \emptyset & \text{otherwise} \end{cases}$$

$$\frac{\texttt{storestack}(\tau, M, \sigma, \delta, R(r)) = \lfloor M' \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{setstack}(r, \tau, \delta).c, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, R, M')}$$

$$\frac{\texttt{loadstack}(\tau, M, \sigma, \delta) = \lfloor v \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{getstack}(\tau, \delta, r).c, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, R\{r \leftarrow v\}, M)}$$

$$\frac{\texttt{loadstack}(\texttt{int}, M, \sigma, F.\texttt{link}) = \lfloor \sigma' \rfloor \quad \texttt{loadstack}(\tau, M, \sigma', \delta) = \lfloor v \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{getparent}(\tau, \delta, r).c, R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, c, R\{r \leftarrow v\}, M)}$$

$$\frac{R(r) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig \quad \texttt{retaddr}(F, c, ra)}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{call}(sig, r).c, R, M) \xrightarrow{\epsilon} \mathcal{C}(\mathcal{F}(F, \sigma, ra, c).\Sigma, Fd, R, M)}$$

$$\frac{\begin{array}{c} R(r) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig \\ \texttt{loadstack}(\texttt{int}, M, \sigma, F.\texttt{link}) = \lfloor \Sigma.\texttt{top}.\sigma \rfloor \quad \texttt{loadstack}(\texttt{int}, M, \sigma, F.\texttt{retaddr}) = \lfloor \Sigma.\texttt{top}.ra \rfloor \end{array}}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{tailcall}(sig, r).c, R, M) \xrightarrow{\epsilon} \mathcal{C}(\Sigma, Fd, R, M)}$$

$$\frac{\texttt{loadstack}(\texttt{int}, M, \sigma, F.\texttt{link}) = \lfloor \Sigma.\texttt{top}.\sigma \rfloor \quad \texttt{loadstack}(\texttt{int}, M, \sigma, F.\texttt{retaddr}) = \lfloor \Sigma.\texttt{top}.ra \rfloor}{G \vdash \mathcal{S}(\Sigma, F, \sigma, \texttt{return}.c, R, M) \xrightarrow{\epsilon} \mathcal{R}(\Sigma, R, M)}$$

$$\frac{\begin{array}{c} \texttt{alloc}(M, F.\texttt{stack\_low}, F.\texttt{stack\_high}) = (b, M_1) \quad \sigma = \texttt{ptr}(b, F.\texttt{stack\_low}) \\ \texttt{storestack}(\texttt{int}, M_1, \sigma, F.\texttt{link}, \Sigma.\texttt{top}.\sigma) = \lfloor M_2 \rfloor \\ \texttt{storestack}(\texttt{int}, M_2, \sigma, F.\texttt{retaddr}, \Sigma.\texttt{top}.ra) = \lfloor M_3 \rfloor \end{array}}{G \vdash \mathcal{C}(\Sigma, \texttt{internal}(F), R, M) \xrightarrow{\epsilon} \mathcal{S}(\Sigma, F, \sigma, F.\texttt{code}, R, M_3)}$$

Fig. 15 Semantics of Mach (selected rules).

relevant in the next compilation pass (generation of assembly code), but it is convenient to reflect it in the semantics of Mach. To this end, the semantics is parameterized by a predicate $\texttt{retaddr}(F, c, ra)$ that relates this return address ra with the caller's function F and the code sequence c that immediately follows the `call` instruction. In section 14.2, we will see how to define this predicate in a way that accurately characterizes the return address. The rules for `return` and `tailcall` contain premises that check that the values contained in the `retaddr` and `link` locations of the activation record were preserved during the execution of the function.

12.2 Code transformation

The translation from Linear to Mach proceeds in two steps. First, the Linear code is scanned to determine which stack slots and callee-save registers it uses. Based on this information, the activation record is laid out following the general shape pictured in figure 16. The total size of the record and the byte offsets for each of its areas are determined. From these offsets, we can define a function Δ mapping callee-save registers, `local` and `outgoing` stack slots to byte offsets, as suggested in figure 16. (Note that these offsets are negative, while positive offsets within the activation frame correspond

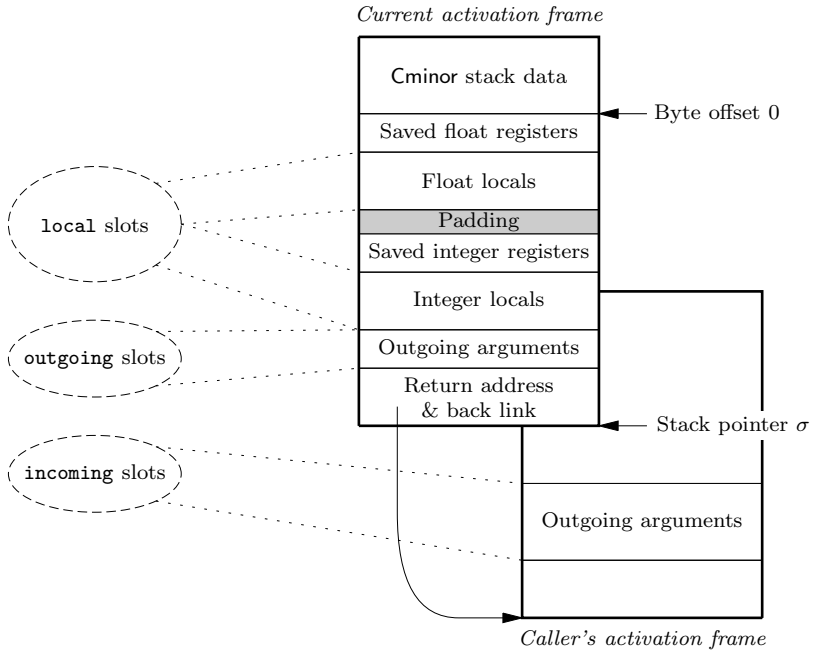

Fig. 16 Layout of Mach activation frames; mapping from Linear stack slots to frame locations.

to Cminor stack data. This choice is compatible with our pointer and memory model, where offsets in pointers are signed integers, and it simplifies the soundness proof.)

The generation of Mach code is straightforward. `getslot` and `setslot` Linear instructions are rewritten as follows:

$$\begin{aligned}
[\![\texttt{getslot}(\texttt{local}(\tau,\delta),r)]\!] &= \texttt{getstack}(\tau,\Delta(\texttt{local}(\tau,\delta)),r)\\
[\![\texttt{getslot}(\texttt{outgoing}(\tau,\delta),r)]\!] &= \texttt{getstack}(\tau,\Delta(\texttt{outgoing}(\tau,\delta)),r)\\
[\![\texttt{getslot}(\texttt{incoming}(\tau,\delta),r)]\!] &= \texttt{getparent}(\tau,\Delta(\texttt{outgoing}(\tau,\delta)),r)\\
[\![\texttt{setslot}(r,\texttt{local}(\tau,\delta))]\!] &= \texttt{setstack}(r,\tau,\Delta(\texttt{local}(\tau,\delta)))\\
[\![\texttt{setslot}(r,\texttt{outgoing}(\tau,\delta))]\!] &= \texttt{setstack}(r,\tau,\Delta(\texttt{outgoing}(\tau,\delta)))
\end{aligned}$$

Moreover, instructions to save and restore the callee-save registers $r_1,\ldots,r_n$ used in the function are inserted as function epilogue and prologue:

$$\begin{aligned}
[\![\texttt{return}]\!] &= \ldots\texttt{getstack}(\tau(r_i),\Delta(r_i),r_i)\ldots\texttt{return}\\
[\![F]\!] &= \{\texttt{code} = \ldots\texttt{setstack}(r_i,\tau(r_i),\Delta(r_i))\ldots[\![F.\texttt{code}]\!];\ldots\}
\end{aligned}$$

Moreover, the translation of a function $[\![F]\!]$ must fail if the "frame" part of the activation record (everything except the Cminor stack data) is bigger than 2^{31} bytes. Indeed, in this case the signed integer offsets used to access locations within the activation record would overflow, making it impossible to access some frame components.

12.3 Semantic preservation

While the code transformation outlined above is simple, its proof of correctness is surprisingly difficult, mostly because it entails reasoning about memory separation properties (between areas of the activation record and between different activation records). To manage this complexity, we broke the proof in two sub-proofs, using an alternate semantics for Mach to connect them.

In this alternate semantics, the "frame" part of activation records does not reside in memory; instead, its contents are recorded separately in a component Φ of regular states. This environment Φ maps (type, byte offset) pairs to values, taking overlap into account in the style of the maps L from locations to values introduced in section 8.1. Each function activation has its own frame environment Φ, saved and restored from the call stack Σ. In the alternate semantics for Mach, the `getstack` and `setstack` instructions are reinterpreted as accesses and updates to Φ, and `getparent` instructions as accesses to $\Sigma.\texttt{top}.\Phi$.

The first part of the proof shows a simulation diagram of the "plus" kind between executions of the original Linear code and executions of the generated Mach code using the alternate semantics outlined above. Thanks to the alternate semantics, memory states are identical between pairs of matching Linear and Mach states. The main invariant to be enforced is agreement between, on the Linear side, the location states L and $\Sigma.\texttt{top}.L$ of the current and calling functions and, on the Mach side, the register state R and the frame states Φ and $\Sigma'.\texttt{top}.\Phi$. This agreement captures the following conditions:

- $L(r) = R(r)$ for all registers r;
- $L(s) = \Phi(\tau, \Delta(s))$ for all `local` and `outgoing` stack slots s of type τ used in the current function;
- $L(s) = \Sigma'.\texttt{top}.\Phi(\tau, \Delta(s))$ for all `incoming` slots s of type τ used in the current function;
- $\Sigma.\texttt{top}.L(r) = \Phi(\tau, \Delta(r))$ for all callee-save registers r of type τ used in the current function;
- $\Sigma.\texttt{top}.L(r) = R(r)$ for all callee-save registers r not used in the current function.

The preservation of agreement during execution steps follows mainly from separation properties between the various areas of activation records.

The second part of the proof shows a lock-step simulation result between executions of Mach programs that use the alternate and standard semantics, respectively. Here, the memory states M (of the alternate semantics) and M' (of the standard semantics) differ: for each activation record b, the block $M'(b)$ is larger than the block $M(b)$ (because it contains additional "frame" data), but the two blocks have the same contents on byte offsets valid for $M(b)$ (these offsets correspond to the Cminor stack data). We capture this connection between memory states by the "memory extension" ordering $M \leq M'$:

$$M \leq M' \quad \stackrel{\text{def}}{=} \quad \forall \kappa, b, \delta, v,\ \texttt{load}(\kappa, M, b, \delta) = \lfloor v \rfloor \Longrightarrow \texttt{load}(\kappa, M', b, \delta) = \lfloor v \rfloor$$

Leroy and Blazy [59, section 5.2] study the properties of this relation between memory states and shows that it commutes nicely with the `store`, `alloc` and `free` operations over memory states.

Another invariant that we must maintain is that the contents of the block $M'(b)$ agree with the frame state Φ of the alternate Mach semantics:

$$\begin{aligned} F, M, M' \models \Phi \sim b, \delta \quad \stackrel{\text{def}}{=} \quad & b \text{ is valid in } M \wedge \mathcal{L}(M, b) = 0 \\ & \wedge\ \mathcal{L}(M', b) = F.\texttt{stack\_low} \wedge \mathcal{H}(M', b) \geq 0 \\ & \wedge\ \delta = F.\texttt{stack\_low} \bmod 2^{32} \\ & \wedge\ \forall \tau, n, \quad F.\texttt{stack\_low} \leq n \wedge n + |\tau| \leq 0 \Longrightarrow \\ & \qquad \texttt{load}(\kappa(\tau), M', b, n) = \lfloor \Phi(\tau, n) \rfloor \end{aligned}$$

The invariant between alternate Mach states and standard Mach states is, then, of the following form:

$$\frac{\Sigma \sim \Sigma' \qquad F, M, M' \models \Phi \sim \sigma, \delta \qquad M \leq M' \qquad \Sigma' \prec \sigma}{\mathcal{S}(\Sigma, F, \texttt{ptr}(\sigma, \delta), c, R, \Phi, M) \sim \mathcal{S}(\Sigma', F, \texttt{ptr}(\sigma, \delta), c, R, M')}$$

The notation $\Sigma' \prec \sigma$ means that the stack blocks $\sigma_1, \ldots, \sigma_n$ appearing in the call stack Σ' are such that $\sigma > \sigma_1 > \ldots > \sigma_n$ and are therefore pairwise distinct. To complete the proof of simulation between the alternate and standard semantics for Mach, we need to exploit well-typedness properties. The frame environments Φ used in the alternate semantics of Mach satisfy the classic "good variable" property $\Phi\{(\tau, \delta) \leftarrow v\}(\tau, \delta) = v$ regardless of whether the value v matches the claimed type τ. However, once frames are mapped to memory locations, writing e.g. a float value with memory quantity `int32` and reading it back with the same memory quantity results in `undef`, not the stored value. More precisely:

Lemma 8 *Assume* $\texttt{storestack}(\tau, M, \sigma, \delta, v) = \lfloor M' \rfloor$. *Then,* $\texttt{loadstack}(\tau, M', \sigma, \delta) = \lfloor v \rfloor$ *if and only if* $v : \tau$ *(as defined in section 3.2).*

Therefore, we need to exploit the well-typedness of the Mach code in a trivial `int`-or-`float` type system in the style of section 4.3 to ensure that the values v stored in a stack location of type τ always semantically belong to type τ. We say that a Mach alternate execution state S is well typed if

- all functions and code sequences appearing in S are well typed;
- all abstract frames Φ appearing in S are such that $\forall \tau, \delta,\ \Phi(\tau, \delta) : \tau$;
- all register states R appearing in S are such that $\forall r,\ R(r) : \tau(r)$.

We can then prove a "subject reduction" lemma showing that well-typedness is preserved by transitions.

Lemma 9 *If* $G \vdash S \xrightarrow{t} S'$ *in the* Mach *abstract semantics and* S *is well typed, then* S' *is well typed.*

Combining this well-typedness property with the invariant between alternate and standard Mach states, we prove the following lock-step simulation result between the two Mach semantics.

Lemma 10 *If* $G \vdash S_1 \xrightarrow{t} S_1'$ *in the* Mach *abstract semantics and* S_1 *is well typed and* $S_1 \sim S_2$, *there exists* S_2' *such that* $G \vdash S_2 \xrightarrow{t} S_2'$ *in the standard semantics and* $S_1' \sim S_2'$.

Semantic preservation for the compiler pass that constructs activation records then follows from the two sub-proofs of simulation outlined above.

Machine instructions:

`add`	`blr`	`fcmpu`	`lfsx`	`nand`	`stfdx`
`addi`	`bt`	`fdiv`	`lha`	`nor`	`stfs`
`addis`	`cmplw`	`fmadd`	`lhax`	`or`	`stfsx`
`addze`	`cmplwi`	`fmr`	`lhz`	`orc`	`sth`
`and.`	`cmpw`	`fmsub`	`lhzx`	`ori`	`sthx`
`andc`	`cmpwi`	`fmul`	`lwz`	`oris`	`stw`
`andi.`	`cror`	`fneg`	`lwzx`	`rlwinm`	`stwx`
`andis.`	`divw`	`frsp`	`mfcr`	`slw`	`subfc`
`b`	`divwu`	`fsub`	`mflr`	`sraw`	`subfic`
`bctr`	`eqv`	`lbz`	`mr`	`srawi`	`xor`
`bctrl`	`extsb`	`lbzx`	`mtctr`	`srw`	`xori`
`bf`	`extsh`	`lfd`	`mtlr`	`stb`	`xoris`
`bl`	`fabs`	`lfdx`	`mulli`	`stbx`	
`bs`	`fadd`	`lfs`	`mullw`	`stfd`	

Macro-instructions:

`allocframe`, `freeframe`: allocation and deallocation of a stack frame
`lfi`: load a floating-point constant in a float register
`fcti`, `fctiu`: conversion from floats to integers
`ictf`, `iuctf`: conversion from integers to floats

Fig. 17 The subset of PowerPC instructions used in Compcert.

13 The output language: PowerPC assembly language

The target language for our compiler is PPC, an abstract syntax for a subset of the PowerPC assembly language [47], comprising 82 of the 200+ instructions of this processor, plus 7 macro-instructions. The supported instructions are listed in figure 17.

13.1 Syntax

The syntax of PPC has the following shape:

Integer registers: $r_i ::= \texttt{R0} \mid \texttt{R1} \mid \ldots \mid \texttt{R31}$

Float registers: $r_f ::= \texttt{F0} \mid \texttt{F1} \mid \ldots \mid \texttt{F31}$

Condition bits: $r_c ::= \texttt{CR0} \mid \texttt{CR1} \mid \texttt{CR2} \mid \texttt{CR3}$

Constants: $cst ::= n \mid \texttt{lo16}(id + \delta) \mid \texttt{hi16}(id + \delta)$

Instructions:
$$\begin{aligned} i ::= {} & \texttt{label}(lbl) \mid \texttt{bt}(r_c, lbl) \\ & \mid \texttt{add}(r_i, r'_i, r''_i) \mid \texttt{addi}(r_i, r'_i, cst) \\ & \mid \texttt{fadd}(r_f, r'_f, r''_f) \mid \ldots \end{aligned}$$

Internal functions: $F ::= i^*$

PPC is an assembly language, not a machine language. This is apparent in the use of symbolic labels in branch instructions such as `bt`, and in the use of symbolic constants $\texttt{lo16}(id + \delta)$ and $\texttt{hi16}(id + \delta)$ as immediate operands of some instructions. (These constants, resolved by the linker, denote the low-order and high-order 16 bits of the memory address of symbol id plus offset δ.)

Moreover, PPC features a handful of macro-instructions that expand to canned sequences of actual instructions during pretty-printing of the abstract syntax to concrete assembly syntax. These macro-instructions include allocation and deallocation of the stack frame (mapped to arithmetic on the stack pointer register), conversions between integers and floats (mapped to memory transfers and complicated bit-level manipulations of IEEE floats), and loading of a floating-point literal (mapped to a load from a memory-allocated constant). The reason for treating these operations as basic instructions is that the memory model and the axiomatization of IEEE float arithmetic that we use are too abstract to verify the correctness of the corresponding canned instruction sequences. (For example, our memory model cannot express that two abstract stack frames are adjacent in memory.) We leave this verification to future work, but note that these canned sequences of instructions are identical to those used by GCC and other PowerPC compilers and therefore have been tested extensively.

13.2 Semantics

Program states in PPC are pairs $S ::= (R, M)$ of a memory state M and a register state R associating values to the processor registers that we model, namely integer registers r_i, floating-point registers r_f, bits 0 to 3 of the condition register `CR`, the program counter `PC`, and the special "link" and "counter" registers `LR` and `CTR`.

The core of PPC's operational semantics is a transition function $T(i, S) = \lfloor S' \rfloor$ that determines the state S' after executing instruction i in initial state S. In particular, the program counter `PC` is incremented (for fall-through instructions) or set to the branch target (for branching instructions). We omit the definition of T in this article, as it is a very large case analysis, but refer the reader to the Coq development for more details. The semantics of PPC, then, is defined by just two transition rules:

$$\frac{R(\texttt{PC}) = \texttt{ptr}(b, n) \qquad G(b) = \lfloor \texttt{internal}(c) \rfloor \qquad c\#n = \lfloor i \rfloor \qquad T(i, (R, M)) = \lfloor (R', M') \rfloor}{G \vdash (R, M) \xrightarrow{\epsilon} (R', M')}$$

$$\frac{R(\texttt{PC}) = \texttt{ptr}(b, 0) \qquad G(b) = \lfloor \texttt{external}(Fe) \rfloor \qquad \texttt{extcall\_arguments}(R, M, Fe.\texttt{sig}) = \lfloor \vec{v} \rfloor \qquad Fe(\vec{v}) \stackrel{t}{\Rightarrow} v \qquad R' = R\{\texttt{PC} \leftarrow R(\texttt{LR}),\ \texttt{extcall\_result}(Fe.\texttt{sig}) \leftarrow v\}}{G \vdash (R, M) \xrightarrow{t} (R', M)}$$

The first rule describes the execution of one PPC instruction within an internal function. The notation $c\#n$ denotes the n^{th} instruction in the list c. The first three premises model abstractly the act of reading and decoding the instruction pointed to by the program counter `PC`. For simplicity, we consider that all instructions occupy one byte in memory, so that incrementing the program counter corresponds to branching to the next instruction. It would be easy to account for variable-length instructions (e.g. 4 bytes for regular instructions, 0 for labels, and $4n$ bytes for macro-instructions). The second rule describes the big-step execution of an invocation of an external function. The `extcall_arguments` function extracts the arguments to the call from the registers and stack memory locations prescribed by the signature of the external call; likewise, `extcall_result` denotes the register where the result must be stored. Conventionally,

the address of the instruction following the call is found in register `LR`; setting `PC` to $R(\texttt{LR})$ therefore returns to the caller.

13.3 Determinism

Determinism of the target language plays an important role in the general framework described in section 2. It is therefore time to see whether the semantics of PPC is deterministic. In the general case, the answer is definitely "no": in the rule for external function calls, the result value v of the call is unconstrained and can take any value, resulting in different executions for the same PPC program. However, this is a form of external nondeterminism: the choice of the result value v is not left to the program, but is performed by the "world" (operating system, user interaction, ...) in which the program executes. As we now formalize, if the world is deterministic, so is the semantics of PPC. A deterministic world is modeled as a partial function W taking the identifier of an external call and its argument values and returning the result value of the call as well as an updated world W'. A finite trace t or infinite trace T is legal in a world W, written $W \models t$ or $W \models T$, if the result values of external calls recorded in the trace agree with what the world W imposes:

$$W \models \epsilon \qquad \frac{W(id, \vec{v}) = \lfloor v, W' \rfloor \qquad W' \models t}{W \models id(\vec{v} \mapsto v).t}$$

$$\frac{W(id, \vec{v}) = \lfloor v, W' \rfloor \qquad W' \models T}{W \models id(\vec{v} \mapsto v).T}$$

We extend this definition to program behaviors in the obvious way:

$$\frac{W \models t}{W \models \texttt{converges}(t, n)} \qquad \frac{W \models T}{W \models \texttt{diverges}(T)} \qquad \frac{W \models t}{W \models \texttt{goeswrong}(t)}$$

We could expect that a PPC program has at most one behavior B that is legal in a deterministic initial world W. This is true for terminating behaviors, but for diverging behaviors a second source of apparent nondeterminism appears, caused by the coinductive definition of the infinite closure relation $G \vdash S \xrightarrow{T} \infty$ in section 3.5. Consider a program P that diverges silently, such as an infinite empty loop. According to the definitions in section 3.5, this program has behaviors $\texttt{diverges}(T)$ for any finite or infinite trace T, not just $T = \epsilon$ as expected. Indeed, no finite observation of the execution of P can disprove the claim that it executes with a trace $T \neq \epsilon$. However, using classical logic (the axiom of excluded middle), it is easy to show that the set of possible traces admits a minimal element for the prefix ordering between traces T. This minimal trace is infinite if the program is *reactive* (performs infinitely many I/O operations separated by finite numbers of internal computation steps) and finite otherwise (if the program eventually loops without performing any I/O). By restricting observations to legal behaviors with minimal traces, we finally obtain the desired determinism property for PPC.

Theorem 7 *Let P be a* PPC *program, W be a deterministic initial world, and $P \Downarrow B$ and $P \Downarrow B'$ be two executions of P. If $W \models B$ and $W \models B'$ and moreover the traces of B and B' are minimal, then $B' = B$ up to bisimilarity of infinite traces.*

14 Generation of PowerPC assembly language

14.1 Code generation

The final compilation pass of Compcert translates from Mach to PPC by expanding Mach instructions into canned sequences of PPC instructions. For example, a Mach conditional branch $\texttt{cond}(\mathit{cond}, \vec{r}, l_{\mathit{true}})$ becomes a `cmplw`, `cmplwi`, `cmpw`, `cmpwi` or `fcmp` instruction that sets condition bits, followed in some cases by a `cror` instruction to merge two condition bits, followed by a `bt` or `bf` conditional branch. Moreover, Mach registers are injected into PPC integer or float registers.

The translation deals with various idiosyncrasies of the PowerPC instruction set, such as the limited range for immediate arguments to integer arithmetic operations, and the fact that register `R0` reads as 0 when used as argument to some instructions. Two registers (`R2` and `F13`) are reserved for these purposes. The translation is presented as a number of "smart constructor" functions that construct and combine sequences of PPC operations. To give the flavor of the translation, here are the smart constructors for "load integer immediate" and "add integer immediate". The functions $\texttt{low}(n)$ and $\texttt{high}(n)$ compute the signed 16-bit integers such that $n = \texttt{high}(n) \times 2^{16} + \texttt{low}(n)$.

$$\texttt{loadimm}(r, n) = \begin{cases} \texttt{addi}(r, \texttt{R0}, n) & \text{if } \texttt{high}(n) = 0; \\ \texttt{addis}(r, \texttt{R0}, \texttt{high}(n)) & \text{if } \texttt{low}(n) = 0; \\ \texttt{addis}(r, \texttt{R0}, \texttt{high}(n)); \texttt{ori}(r, r, \texttt{low}(n)) & \text{otherwise} \end{cases}$$

$$\texttt{addimm}(r_d, r_s, n) = \begin{cases} \texttt{loadimm}(\texttt{R2}, n); \texttt{add}(r_d, r_s, \texttt{R2}) & \text{if } r_d = \texttt{R0} \text{ or } r_s = \texttt{R0}; \\ \texttt{addi}(r_d, r_s, n) & \text{if } \texttt{high}(n) = 0; \\ \texttt{addis}(r_d, r_s, \texttt{high}(n)) & \text{if } \texttt{low}(n) = 0; \\ \texttt{addis}(r_d, r_s, \texttt{high}(n)); \texttt{addi}(r_d, r_d, \texttt{low}(n)) & \text{otherwise} \end{cases}$$

Just as the generation of Mach code must fail if the activation record is too large to be addressed by machine integers (section 12.2), the PPC generator must fail if the translation of a function contains 2^{31} or more instructions, since it would then be impossible to address some of the instructions via a signed 32-bit offset from the beginning of the function.

14.2 Semantic preservation

Semantic preservation for PPC generation is proved using a simulation diagram of the "option" type. The two main invariants to be preserved are:

1. The `PC` register contains a pointer $\texttt{ptr}(b, \delta)$ that corresponds to the Mach function F and code sequence c currently executing:

$$F, c \sim \texttt{ptr}(b, \delta) \quad \stackrel{\text{def}}{=} \quad G(b) = \lfloor \texttt{internal}(F) \rfloor \ \wedge \ [\![c]\!] = \texttt{suffix}([\![F]\!], \delta)$$

2. The Mach register state R and stack pointer σ agree with the PPC register state R':

$$R, \sigma \sim R' \quad \stackrel{\text{def}}{=} \quad R'(\texttt{R1}) = \sigma \ \wedge \ \forall r, R(r) = R'(\overline{r})$$

   where $\overline{r}$ denotes the PPC register associated with the Mach register r. (Conventionally, the `R1` register is used as the stack pointer.)

More precisely, matching between a Mach execution state and a PPC execution state is defined as follows:

$$\frac{\Sigma\ \texttt{wf} \qquad F, c \sim R'(\texttt{PC}) \qquad R, \sigma \sim R'}{\mathcal{S}(\Sigma, F, \sigma, c, R, M) \sim (R', M)}$$

$$\frac{\Sigma\ \texttt{wf} \qquad R'(\texttt{PC}) = \texttt{ptr}(b, 0) \qquad G(b) = \lfloor Fd \rfloor \qquad R'(\texttt{LR}) = \Sigma.\texttt{top}.ra \qquad R, \Sigma.\texttt{top}.\sigma \sim R'}{\mathcal{C}(\Sigma, Fd, R, M) \sim (R', M)}$$

$$\frac{\Sigma\ \texttt{wf} \qquad R'(\texttt{PC}) = \Sigma.\texttt{top}.ra \qquad R, \Sigma.\texttt{top}.\sigma \sim R'}{\mathcal{R}(\Sigma, R, M) \sim (R', M)}$$

The invariant Σ wf over Mach call stacks is defined as $F, c \sim ra$ for all $\mathcal{F}(F, \sigma, ra, c) \in \Sigma$. The case for call states reflects the convention that the caller saves its return address in register LR before jumping to the first instruction of the callee. As mentioned in section 12.1, the Mach semantics is parameterized by an oracle $\texttt{retaddr}(F, c, ra)$ that guesses the code pointer ra pointing to the PPC code corresponding to the translation of Mach code c within function F. We construct a suitable oracle by noticing that the translation of a Mach code $i_1 \dots i_n$ is simply the concatenation $[\![i_1]\!] \dots [\![i_n]\!]$ of the translations of the instructions. Therefore, the offset of the return address ra is simply the position of the suffix $[\![c]\!]$ of $[\![F]\!]$. It is always uniquely defined if c is a suffix of $F.\texttt{code}$, which the Mach semantics guarantees. The simulation diagram is of the "star" kind because Mach transitions from return states $\mathcal{R}$ to regular states $\mathcal{S}$ correspond to zero transitions in the generated PPC code. The absence of infinite stuttering is trivially shown by associating measure 1 to return states and measure 0 to regular and call states. The proof of the simulation diagram is long because of the high number of cases to consider but presents no particular difficulties. To reason more easily about the straight-line execution of a sequence of non-branching PPC instructions, we make heavy use of the following derived execution rule:

$$\frac{\begin{array}{c} R_0(\texttt{PC}) = \texttt{ptr}(b, \delta) \qquad G(b) = \lfloor \texttt{internal}(c) \rfloor \qquad \texttt{suffix}(c, \delta) = i_1 \dots i_n.c' \qquad n > 0 \\ \text{for all } k \in \{1, \dots, n\},\ T(i_k, (R_{k-1}, M_{k-1})) = \lfloor (R_k, M_k) \rfloor \text{ and } R_k(\texttt{PC}) = R_{k-1}(\texttt{PC}) + 1 \end{array}}{G \vdash (R_0, M_0) \xrightarrow{\epsilon}{}^{+} (R_n, M_n)}$$

15 The Coq development

We now discuss the Coq implementation of the algorithms and proofs described in this paper.

15.1 Specifying in Coq

The logic implemented by Coq is the Calculus of Inductive and Coinductive Constructions (CICC), a very powerful constructive logic. It supports equally well three familiar styles of writing specifications: by functions and pattern-matching, by inductive or coinductive predicates representing inference rules, and by ordinary predicates in first-order logic. All three styles are used in the Compcert development, resulting

in specifications and statements of theorems that remain quite close to what can be found in programming language research papers.

CICC also features higher-order logic, dependent types, a hierarchy of universes to enforce predicativity, and an ML-style module system. These advanced logical features are used sparingly in our development: higher-order functions and predicates are used for iterators over data structures and to define generic closure operations; simple forms of dependent types are used to attach logical invariants to some data structures and some monads (see section 6.4); coinduction is used to define and reason about infinite transition sequences; parameterized modules are used to implement the generic dataflow solvers of section 7.1. Most of our development is conducted in first-order logic, however.

Two logical axioms (not provable in Coq) are used: function extensionality (if $\forall x,\ f(x) = g(x)$, then the functions f and g are equal) and proof irrelevance (any two proof terms of the same proposition P are equal). It might be possible to avoid using these two axioms, but they make some proofs significantly easier. The proofs of the semantic preservation theorems are entirely constructive. The only place where classical logic is needed, under the form of the axiom of excluded middle, is to show the existence of minimal traces for infinite executions (see section 13.3). This set of three axioms is consistent with the predicative version of Coq's logic that we use.

15.2 Proving in Coq

While Coq proofs can be checked a posteriori in batch mode, they are developed interactively using a number of tactics as elementary proof steps. The sequence of tactics used constitutes the proof script. Building such scripts is surprisingly addictive, in a videogame kind of way, but reading them is notoriously difficult. We were initially concerned that adapting and reusing proof scripts when specifications change could be difficult, forcing many proofs to be rewritten from scratch. In practice, careful decomposition of the proofs in separate lemmas enabled us to reuse large parts of the development even in the face of major changes in the semantics, such as switching from the "mixed-step" semantics described in [57] to the small-step transition semantics described in this paper.

Our proofs frequently use the basic proof automation facilities provided by Coq, mostly `eauto` (Prolog-style resolution), `omega` (Presburger arithmetic) and `congruence` (a decision procedure for ground equalities with uninterpreted symbols). However, these tactics do not combine automatically, and significant manual massaging of the goals is necessary before they apply. The functional induction and functional inversion mechanisms of Coq 8.1 [6] helped reason about functions defined by complex pattern-matching.

Coq also provides a dedicated `Ltac` language for users to define their own tactics. We used this facility occasionally, for instance to define a "monadic inversion" tactic that recursively simplifies hypotheses of the form $(\texttt{do}\ x\ \texttt{<-}\ a; b)\ s = \texttt{OK}(s', r)$ into $\exists s_1.\ \exists\ x.\ a\ s = \texttt{OK}(s_1, x) \wedge b\ s_1 = \texttt{OK}(s', r)$. There is no doubt that a Coq expert could have found more opportunities for domain-specific tactics and could have improved significantly our proof scripts.

As mentioned earlier, most of the development is conducted in first-order logic, suggesting the possibility of using automated theorem provers such as SMT solvers. Preliminary experiments with using SMT solvers to prove properties of the Compcert

	Code (in Coq)	Code (in Caml)	Specifications	Statements	Proof scripts	Directives	Total
Data structures, proof auxiliaries	413	83	-	471	1127	336	2430
Integers, floats, values, memory model	850	21	-	1196	2819	119	5005
Common syntactic and semantic definitions	122	-	111	439	775	517	1964
Cminor semantics	-	-	563	39	186	85	873
CminorSel semantics, operators	-	-	639	88	221	85	1033
Instruction selection, reassociation	838	-	-	399	755	82	2074
RTL semantics	-	-	403	62	198	69	732
RTL generation	413	55	-	931	1237	161	2797
Dataflow optimizations over RTL	1162	-	-	596	1586	282	3626
Calling conventions	117	-	-	127	315	37	596
LTL semantics	0	0	414	48	129	40	631
Register allocation	303	544	0	564	975	190	2576
LTLin semantics	0	0	243	5	10	27	285
Branch tunneling and code linearization	137	52	0	349	674	97	1309
Linear semantics	0	0	280	3	4	27	314
Spilling and reloading	333	0	0	763	1669	171	2936
Mach semantics	-	-	545	38	106	73	762
Layout of the activation record	236	-	-	801	1510	203	2750
PPC semantics	-	-	520	6	7	29	562
PPC generation	424	-	-	832	1904	171	3331
Compiler driver, PPC printer	34	290	-	223	336	101	984
Total	5382	1045	3718	7980	16543	2902	37570
	14%	3%	10%	21%	44%	8%	

Fig. 18 Size of the development (in non-blank, non-comment lines of code).

memory model [59] indicate that many but not all of the lemmas can be proved automatically. While fully automated verification of a program like Compcert appears infeasible with today's technology, we expect that our interactive proof scripts would shrink significantly if Coq provided a modern SMT solver as one of its tactics.

15.3 Size of the development

The size of the Coq development can be estimated from the line counts given in figure 18. The whole development represents approximately 37000 lines of Coq (excluding comments and blank lines) plus 1000 lines of code directly written in Caml. The overall effort represents approximately 2 person-years of work.

The Coq datatype and function definitions that implement the compiler itself (column "Code in Coq" in figure 18) account for 14% of the source. In other terms, the Coq verification is about 6 times larger than the program being verified. The remaining 86% comprise 10% of specifications (mostly, the operational semantics for the source, target and intermediate languages), 21% of statements of lemmas, theorems and supporting definitions, 44% of proof scripts and 8% of directives, module declarations, and custom proof tactics.

The sizes of individual passes are relatively consistent: the most difficult passes (RTL generation, register allocation, spilling and reloading, and layout of the activa-

tion record) take between 2000 and 3000 lines each, while simpler passes (constant propagation, CSE, tunneling, linearization) take less than 1500 lines each. One outlier is the PPC code generation pass which, while conceptually simple, involves large definitions and proofs by case analysis, totaling more than 3300 lines. Among the supporting libraries, the formalizations of machine integer arithmetic and of the memory model are the largest and most difficult, requiring 1900 and 2300 lines respectively.

Checking all the proofs in the development takes about 7.5 minutes of CPU time on a 2.4 GHz Intel Core 2 processor equipped with 4 Gb of RAM and running 64-bit Linux. The version of Coq used is 8.1pl3. Parallel `make` with 2 cores results in a wall-clock time of 4.5 minutes.

16 Experimental results

16.1 Extracting an executable compiler

As mentioned in the Introduction, the verified parts of the Compcert compiler are programmed directly in Coq, then automatically translated to executable Caml code using Coq's extraction facility [61, 62]. To obtain an executable compiler, this extracted Caml code is combined with:

- A compiler front-end, translating the Clight subset of C to Cminor. This front-end is itself extracted from a Coq development. An earlier version of this front-end compiler is described in [15].
- A parser for C that generates Clight abstract syntax. This parser is built on top of the CIL library [77].
- Hand-written Caml implementations of the heuristics that we validate a posteriori (graph coloring, RTL type reconstruction, etc), of the pretty-printer for PowerPC assembly code, and of a `cc`-style compiler driver.

The resulting compiler runs on any platform supported by Caml and generates PowerPC code that runs under MacOS X. While the soundness proof for Compcert does not account for separate compilation and assumes that whole programs are compiled at once, the compiler can be used to separately compile C source files and link them with precompiled libraries, which is convenient for testing. (The calling conventions implemented by Compcert are compatible enough with the standard PowerPC ABI to support this.)

Program extraction performs two main tasks. First, it eliminates the parts of Coq terms that have no computational content, by a process similar to program slicing. For instance, if a data structure carries a logical invariant, every instance of this structure contains a proof term showing that the invariant is satisfied. This subterm does not contribute to the final result of the program, only to its correctness, and is therefore eliminated by extraction. The second task is to bridge the gap between Coq's rich type system and Caml's simpler Hindley-Milner type system. Uses of first-class polymorphism or general dependent types in Coq can lead to programs that are not typeable in Caml; extraction works around this issue by inserting unsafe Caml coercions, locally turning off Caml's type checking. This never happens in Compcert, since the source Coq code is written in pedestrian ML style, using only Hindley-Milner types.

Generally speaking, the Caml code extracted from the Compcert development looks like what a Caml programmer would write if confined to the purely functional subset

Test program	`gcc -O0`	Comp-cert	`gcc -O1`	`gcc -O2`
AES cipher	6.00s	39.2%	29.4%	25.5%
Almabench	1.22s	90.4%	83.5%	84.5%
Arithmetic coding	8.00s	52.6%	39.8%	38.8%
Binary trees	14.08s	74.4%	78.6%	78.1%
Fannkuch	2.23s	31.6%	27.7%	25.6%
FFT	0.52s	85.1%	82.6%	81.5%
K-nucleotide	0.21s	75.3%	74.4%	74.9%
Lempel-Ziv	79.43s	20.2%	14.5%	14.5%
Lempel-Ziv-Welch	26.67s	25.5%	22.5%	21.5%
Mandelbrot	8.93s	33.6%	28.3%	29.6%
N-body	34.00s	37.7%	44.9%	31.6%
Number sieve	1.06s	64.0%	61.7%	60.1%
Quicksort	1.36s	41.0%	35.7%	36.0%
Ray tracer	11.57s	47.6%	41.7%	39.9%
SHA1 hash	4.17s	23.7%	24.0%	22.1%
Spectral test	17.01s	55.2%	54.9%	53.3%
Virtual machine	42.50s	24.4%	37.1%	35.0%
Geometric mean		43.8%	40.8%	38.9%

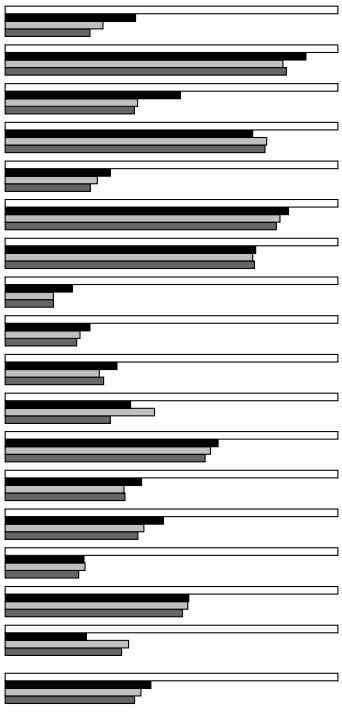

Fig. 19 Execution times of compiled code. Times are given relative to those obtained with `gcc -O0`. Lower percentages and shorter bars mean faster.

of the language. There are two exceptions. The first is related to the handling of default cases in Coq pattern-matching. Consider a data type with 5 constructors `A` to `E`, and a pattern-matching `(A, A) -> x | (_, _) -> y`. Internally, Coq represents this definition by a complete matching having 5×5 cases, 24 of which are `y`. Extraction does not yet re-factor the default case, resulting in 24 copies of the code `y`. On large data types, this can lead to significant code explosion, which we limited on a case by case basis, often by introducing auxiliary functions.

The other problem with extraction we encountered is the η-expansions that extraction sometimes performs in the hope of exposing more opportunities for program slicing. These expansions can introduce inefficiencies by un-sharing computations. Consider for example a curried function of two arguments x and y that takes x, performs an expensive computation, then returns a function $\lambda y \ldots$. After η-expansion, this expensive computation is performed every time the second argument is passed. We ran into this problem twice in Compcert. Manual patching of the extracted Caml code was necessary to undo this "optimization".

16.2 Benchmarks

Performance of the generated code can be estimated from the timings given in figure 19. Since Compcert accepts only a subset of the C language (excluding variadic functions and `long long` arithmetic types, for instance), standard benchmark suites cannot be used, and we reverted to a small home-grown test suite. The test programs range from 50 to 3000 lines of C code, and include computational kernels (FFT, N-body, etc.), cryptographic primitives (AES, SHA1), text compression algorithms, a virtual machine interpreter, and a ray tracer derived from the ICFP 2000 programming contest. The PowerPC code generated by Compcert was benchmarked against the code generated by GCC version 4.0.1 at optimization levels 0, 1 and 2. (Higher GCC optimization levels make no significant differences on this test suite.) For the purpose of this benchmark, Compcert was allowed to recognize the fused multiply-add and multiply-sub PowerPC instructions. (These instructions are normally not used in Compcert because they produce results different from a multiply followed by an add or sub, but since GCC uses them nonetheless, it is fair to allow Compcert to do so as well.) Measurements were performed on an Apple PowerMac workstation with two 2.0 GHz PowerPC 970 (G5) processors and 6 Gb of RAM, running MacOS 10.4.11.

As the timings in figure 19 show, Compcert generates code that is more than twice as fast as that generated by GCC without optimizations, and competitive with GCC at optimization levels 1 and 2. On average, Compcert code is only 7% slower than `gcc -O1` and 12% slower than `gcc -O2`. The test suite is too small to draw definitive conclusions, but these results strongly suggest that while Compcert is not going to win any prize in high performance computing, the performance of generated programs is adequate for critical embedded code.

Compilation times are higher for Compcert than for GCC but remain acceptable: to compile the 3000-line ray tracer, Compcert takes 4.6 s while `gcc -O1` takes 2.7 s. There are several possible reasons for this slowdown. One is that Compcert proceeds in a relatively high number of passes, each of which reconstructs entirely the representation of the function being compiled. Another is the use of purely functional data structures (balanced trees) instead of imperative data structures (bitvectors, hash tables). We were careful, however, to use functional data structures with good asymptotic complexity (mainly AVL trees and radix-2 trees), which cost only a logarithmic factor compared with imperative data structures.

17 Discussion and perspectives

We now discuss some of the design choices and limitations of our work and outline directions for future work.

17.1 On Cminor as a target language

The Cminor intermediate language was designed to allow relatively direct translation of a large subset of the C language. In particular, the memory model closely matches that of C, and the `block`/`exit` mechanism makes it easy to translate C loops (including `break` and `continue`) in a compositional manner. The C feature that appears most diffi-

cult to support is variadic functions; this is not surprising given that variadic functions account for most of the complexity of function calling conventions in C compilers.

Other features of C could be supported with small extensions to Cminor and the Compcert back-end. For instance, Cminor currently performs all floating-point arithmetic in double precision, but it is planned to add single-precision float operators to better support ISO C's floating-point semantics. Also, large `switch` statements could be compiled more efficiently if multi-way branches (jump tables) were added to RTL and later intermediate languages.

Less obviously, Cminor can also be used as a target language when compiling higher-level source languages. Function pointers and tail-call optimization are supported, enabling the compilation of object-oriented and functional languages. Exceptions in the style of ML, C++ or Java are not supported natively in Cminor but could be encoded (at some run-time cost) either as special return values or using continuation-passing style. For the former approach, it could be worthwhile to add functions with multiple return values to Cminor.

For languages with automatic memory management, Cminor provides no native support for accurate garbage collection: mechanisms for tracking GC roots through register and stack allocation in the style of C-- [81] are not provided and appear difficult to specify and prove correct. However, the Cminor producer can explicitly register GC roots in Cminor stack blocks, in the style of Henderson [42]. Zaynah Dargaye has prototyped a verified front-end compiler from the mini-ML functional language to Cminor that follows this approach [27].

17.2 On retargeting

While some parts of the Compcert back-end are obviously specific to the PowerPC (e.g. generation of assembly language, section 14), most parts are relatively independent of the target processor and could conceivably be reused for a different target. To make this claim more precise, we experimented with retargeting the Compcert back-end for the popular ARM processor. The three aspects of this port that required significant changes in the back-end and in its proof are:

- Reflecting the differences in instruction sets, the types and semantics of machine-specific operators, addressing modes and conditions (section 5.1) change. This impacts the instruction selection pass (section 5) but also the abstract interpretation of these operators performed by constant propagation (section 7.2).
- Calling conventions and stack layout differ. Most differences are easy to abstract over, but the standard ARM calling convention could not be supported: it requires that floats are passed in pairs of integer registers, which our value and memory model cannot express yet. Nonstandard calling conventions, using float registers to pass floats, had to be used instead.
- ARM has fewer registers than PowerPC (16 integer registers and 8 float registers instead of 32 and 32). Consequently, we had to reduce the number of registers reserved to act as temporary registers. This required some changes in the spilling and reloading pass (section 11).

Overall, the port of Compcert and its proof to the ARM processor took about 3 weeks. Three more weeks were needed to revise the modular structure of the Coq development,

separating the processor-specific parts from the rest, adapting the PowerPC and ARM-specific parts so that they have exactly the same interface, and making provisions for supporting other target processors in the future. Among the 37500 lines of the initial development, 28000 (76%) were found to be processor-independent and 8900 (24%) ended up in PowerPC-specific modules; an additional 8800 lines were added to support the ARM processor.

17.3 On optimizations

As mentioned in the Introduction, the main objective for the Compcert project was to prove end-to-end semantic preservation. This led us to concentrate on non-optimizing transformations that are required in a compiler and to spend less time on optimizations that are optional. Many interesting optimizations [73, 1] remain to be proved correct and integrated in Compcert.

In separate work not yet part of Compcert, Jean-Baptiste Tristan verified two additional optimizations: instruction scheduling by list scheduling and trace scheduling [32] and lazy code motion (LCM) [51]. These optimizations are more advanced than those described in section 7, since they move instructions across basic blocks and even across loop boundaries in the case of LCM. In both cases, Tristan used a translation validation approach (see section 2.2.2) where the code transformation is performed by untrusted Caml code, then verified a posteriori using a verifier that is formally proved correct in Coq. In the case of instruction scheduling, validation is performed by symbolic execution of extended basic blocks [93]. For LCM, validation exploits equations between program variables obtained by an available expressions analysis, combined with an anticipability analysis [94]. On these two examples, the verified translation validation approach was effective, resulting in relatively simple semantic correctness proofs that are insensitive to the many heuristics decisions taken by these two optimizations. We conjecture that for several other optimizations, the verified translation validation approach is simpler than proving directly the correctness of the optimization.

Many advanced optimizations are formulated in terms of static single assignment (SSA) representation rather than over classic intermediate representations like those currently used in Compcert. SSA enables more efficient static analyses and sometimes simpler code transformations. A typical SSA-based optimization that interests us is global value numbering [88]. Since the beginning of Compcert we have been considering using SSA-based intermediate languages, but were held off by two difficulties. First, the dynamic semantics for SSA is not obvious to formalize (see [17, 10, 84] for various approaches). Second, the SSA property is global to the code of a whole function and not straightforward to exploit locally within proofs. Functional representations such as A-normal forms could offer some of the benefits of SSA with clearer semantics. In a translation validation setting, it might not be necessary to reason directly over the SSA form: the untrusted optimizations could convert to SSA, use efficient SSA-based algorithm and convert out of SSA; the validator, which is the only part that needs proving, can still use conventional RTL-like representations.

17.4 On memory

Whether to give formal semantics to imperative languages or to reason over pointer-based programs and transformations over these programs, the memory model is a crucial ingredient. The formalization of memory used in Compcert can be extended and refined in several directions.

The first is to add memory allocation and deallocation primitives to Cminor, in the style of C's `malloc` and `free`. Both can be implemented in Cminor by carving sub-blocks out of a large global array, but primitive support for these operations could facilitate reasoning over Cminor programs. Supporting dynamic allocation is easy since it maps directly to the `alloc` function of our memory model. (This model does not assume that allocations and deallocations follow a stack-like discipline.) Dynamic, programmer-controlled deallocation requires more care: mapping it to the `free` operation of our memory model opens up the possibility that a Cminor function explicitly deallocates its own stack block. This could invalidate semantic preservation: if the Cminor function does not use its stack block and does not terminate, nothing wrong happens, but if some of its variables are spilled to memory, the corresponding Mach code could crash when accessing a spilled variable. Explicit deallocation of stack frames must therefore be prevented at the Cminor level, typically by tagging memory blocks as belonging to the stack or to the heap.

Another limitation of our memory model is that it completely hides the byte-level representation of integers and floats in memory. This makes it impossible to express in Cminor some C programming idioms of dubious legality but high practical usefulness such as copying byte-per-byte an arbitrary data structure to another of the same layout (in the style of `memmove`). Doing so in Cminor would fill the destination structure with `undef` values. As mentioned in section 13.1, this feature of our memory model also prevents us from reasoning about machine code that manipulates the IEEE representation of floats at the bit level. As discussed in [59, section 7], a strong reason for hiding byte-level in-memory representations is to ensure that pointer values cannot be forged from integers or floats; this guarantee plays a crucial role in proving semantic preservation for certain memory transformations. A topic for future work is to refine the memory model to obtain the best of both worlds: unforgeable pointers and byte-level access to the representations of integers and floats.

Compared with "real" memory implementations or even with the version of our memory model presented in [59], the memory model currently used in Compcert makes two simplifying assumptions: (1) `free` never fails, therefore allowing repeated deallocation of a given block; (2) `alloc` never fails, therefore modeling an infinite memory. Assumption (1) is not essential, and our proofs extend straightforwardly to showing that the compiler never inserts a double `free` operation. Assumption (2) on infinite memory is more difficult to remove, because in general a compiler does not preserve the stack memory consumption of the program it compiles. It is easy to show that the generated code performs exactly the same dynamic allocations and deallocations as the source program; therefore, heap memory usage is preserved. However, in Compcert, the sizes of stack blocks can increase arbitrarily between Cminor and PPC, owing to the spilling of Cminor variables to the stack described in section 12. A Cminor program that executes correctly within N bytes of stack space can therefore be translated to a PPC program that runs out of stack, significantly weakening the semantic preservation theorems that we proved.

In other words, while heap memory usage is clearly preserved, it seems difficult to prove a bound on stack usage on the source program and expect this resource certification to carry over to compiled code: stack consumption, like execution time, is a program property that is not naturally preserved by compilation. A simpler alternative is to establish the memory bound directly on the compiled code. If recursion and function pointers are not used, which is generally the case for critical embedded software, a simple, verified static analysis over Mach code that approximates the call graph can provide the required bound N on stack usage. We could, then, prove a strong semantic preservation theorem of the form "if the PPC stack is of size at least N, the generated PPC code behaves like the source Cminor program". For programs that use recursion or function pointers, the issue remains open, however.

17.5 On the multiplicity of passes and intermediate languages

The number of compilation passes in Compcert is relatively high by compiler standards, but not shockingly so. The main motivation here was to have passes that do exactly one thing each, but do it well and in a complete manner. Combining several passes together tends to complicate their proofs super-linearly. For example, the first published version of Compcert [57] performed register allocation, reloading, spilling and enforcement of calling conventions all in one pass. Splitting this pass in two (register allocation in section 8, spilling in section 11) resulted in a net simplification of the proofs.

What is more surprising is the high number of intermediate languages involved in Compcert: with the exception of dataflow-based optimizations (constant propagation, CSE, and tunneling), each pass introduces a new intermediate language to use as its target language. Many of these intermediate languages are small variations on one another. Yet we found it necessary to define each intermediate language separately, rather than identifying them with subsets of a small number of more general intermediate languages (like for instance GCC does with its Tree and RTL representations).

The general problem we face is that of transmitting information between compiler passes: what are the properties of its output that a compiler pass guarantees and that later passes can rely on? These guarantees can be positive properties (e.g. "the Mach code generated by the stack layout pass is compatible with treating registers as global variables") but also negative, "don't care" properties (e.g. "the LTL code generated by register allocation does not use temporary registers and is insensitive to modifications of caller-save registers across function calls").

Some of these guarantees can be captured syntactically. For instance, code produced by register allocation never mentions temporary registers. In Compcert, such syntactic guarantees for a compiler pass are enforced either by the abstract syntax of its target language or via an additional inductive predicate on this abstract syntax. As mentioned in section 4.3, all intermediate languages from Cminor to Mach are weakly typed in a `int`-or-`float` type system. The corresponding typing rules are a good place to carry additional restrictions on the syntax of intermediate languages. For example, LTL's typing rules enforce the restriction that all locations used are either non-temporary registers or stack slots of the `local` kind.

Other guarantees are semantic in nature and cannot be expressed by syntactic restriction. One example is the fact that LTL code generated by register allocation does not expect caller-save registers to be preserved across function calls. Such properties need to be reflected in the dynamic semantics for the target language of the considered

pass. Continuing the previous example, the LTL semantics captures the property by explicitly setting caller-save registers to the `undef` value when executing a function call (section 8.1). Later passes can then refine these `undef` values to whatever value is convenient for them (section 11.3).

In other cases, we must guarantee that an intermediate representation not only doesn't care about the values of some locations but actually preserves whatever values they hold. Just setting these locations to `undef` is not sufficient to capture this guarantee. The approach we followed is to anticipate, in the semantics of an intermediate language, the actual values that these locations will take after later transformations. For example, the Linear semantics (section 11.1) anticipates the saving and reloading of callee-save registers performed by Mach code generated by the stack layout pass (section 12). Another instance of this technique is the semantics of Mach that anticipates the return address that will be stored by PPC code in the slot of the activation record reserved for this usage (sections 12.1 and 14.2). While this technique of semantic anticipation was effective in Compcert, it is clearly not as modular as one would like: the semantics of an intermediate language becomes uncomfortably dependent on the effect of later compilation passes. Finding better techniques to capture behaviors of the form "this generated code does not depend on the value of X and guarantees to preserve the value of X" is an open problem.

17.6 Toward machine language and beyond

The Compcert compilation chain currently stops at the level of an assembly language following a Harvard architecture and equipped with a C-like memory model. The next step "down" would be machine language following a von Neumann architecture and representing memory as a finite array of bytes. The main issue in such a refinement is to bound the amount of memory needed for the call stack, as discussed in section 17.4. Once this is done, we believe that the refinement of the memory model can be proved correct as an instance of the memory embeddings studied by Leroy and Blazy [59, section 5.1]. Additionally, symbolic labels must be resolved into absolute addresses and relative displacements, and instructions must be encoded and stored in memory. Based on earlier work such as [70], such a refinement is likely to be tedious but should not raise major difficulties.

The main interest in going all the way to machine language is to connect our work with existing and future hardware verification efforts at the level of instruction set architectures (i.e. machine language) and micro-architectures. Examples of such hardware verifications include the Piton project [69, 70] (from a high-level assembly language to an NDL netlist for a custom microprocessor), Fox's verification of the ARM6 micro-architecture [34], and the VAMP project [14] (from the DLX instruction set to a gate-level description of a processor). Sharing a specification of an instruction set architecture between the verification of a compiler and the verification of a hardware architecture strengthens the confidence we can have in both verifications.

17.7 Toward shared-memory concurrency

Shared-memory concurrency is back into fashion these days and is infamous for raising serious difficulties both with the verification of concurrent source programs and with

the reuse, in a concurrent setting, of languages and compilers designed for sequential execution [18]. An obvious question, therefore, is whether the Compcert back-end and its soundness proof could be extended to account for shared-memory concurrency.

It is relatively easy to give naive interleaving semantics for concurrency in Cminor and the other languages of Compcert, but semantic preservation during compilation obviously fails: if arbitrary data races are allowed in the source Cminor program, the transformations performed by the compiler introduce additional interleavings and therefore additional behaviors not present in the source program. For instance, the evaluation of an expression is an atomic step in the Cminor semantics but gets decomposed into several instructions during compilation. The weakly consistent hardware memory models implemented by today's processors add even more behaviors that cannot be predicted easily by the Cminor semantics.

Our best hope to show a semantic preservation result in a shared-memory concurrent setting is, therefore, to restrict ourselves to race-free source programs that implement a proper mutual exclusion discipline on their memory accesses. A powerful way to characterize such programs is concurrent separation logic [79]. Using this approach, Hobor et al. [43] develop an operational semantics for Concurrent Cminor, an extension of Cminor with threads and locks. This semantics is pseudo-sequential in that threads run sequentially between two operations over locks, and their interleaving is determined by an external *oracle* that appears as a parameter to the semantics. It is conceivable that, for a fixed but arbitrary oracle, the Compcert proofs of semantic preservation would still hold. The guarantees offered by concurrent separation logic would then imply that the pseudo-sequential semantics for the generated PPC code captures all possible actual executions of this code, even in the presence of arbitrary interleavings and weakly-consistent hardware memory. This approach is very promising, but much work remains to be done.

18 Related work

We have already discussed the relations between compiler verification and other approaches to trusted compilation in section 2. Proving the correctness of compilers has been an active research topic for more than 40 years, starting with the seminal work of McCarthy and Painter [66]. Since then, a great many on-paper proofs for program analyses and compiler transformations have been published — too many to survey here. Representative examples include the works of Clemmensen and Oest [24], Chirica and Martin [22], Guttman et al. [39], Müller-Olm [74] and Lacey et al. [52]. We refer the reader to Dave's annotated bibliography [28] for further references. In the following, we restrict ourselves to discussing correctness proofs of compilers that involve mechanized verification.

Milner and Weyhrauch [68] were arguably the first to mechanically verify semantic preservation for a compilation algorithm (the translation of arithmetic expressions to stack machine code), using the Stanford LCF prover. The first mechanized verification of a full compiler is probably that of Moore [69, 70], although for a rather low-level, assembly-style source language.

The Verifix project [36] had goals broadly similar to ours: the construction of mathematically correct compilers. It produced several methodological approaches, but few parts of this project led to machine-checked proofs. The parts closest to the present work are the formal verification (in PVS) of a compiler from a subset of Common Lisp

to Transputer code [30] and of instruction selection by a bottom-up rewrite system [29].

Strecker [91] and Klein and Nipkow [49] verified non-optimizing byte-code compilers from a subset of Java to a subset of the Java Virtual Machine using Isabelle/HOL. They did not address compiler optimizations nor generation of actual machine code. Another verification of a byte-code compiler is that of Grégoire [37], for a functional language.

The Verisoft project [80] is an ambitious attempt at end-to-end formal verification that covers the whole spectrum from application to hardware. The compiler part of Verisoft is the formal verification of a compiler for a Pascal-like language called C0 down to DLX machine code using the Isabelle/HOL proof assistant [53,92,54]. The publications on this verification lack details but suggest a compiler that is much simpler than Compcert and generates unoptimized code.

Li et al. [63,64] describe an original approach to the generation of trusted ARM code where the input language is a subset of the HOL specification language. Compilation is not entirely automatic: the user chooses interactively which transformations to apply, but the system produces formal evidence (a HOL proof term) that the generated ARM code conforms to the HOL specification.

Chlipala [23] developed and proved correct a compiler for simply-typed λ-calculus down to an idealized assembly language. This Coq development cleverly uses dependent types and type-indexed denotational semantics, resulting in remarkably compact proofs. Another Coq verification of a compiler from a simply-typed functional language to an idealized assembly language is that of Benton and Hur [9]. Like Chlipala's, their proof has a strong denotational semantics flavor; it builds upon the concepts of step-indexed logical relations and biorthogonality. It is unclear yet whether such advanced semantic techniques can be profitably applied to low-level, untyped languages such as those considered in this paper, but this is an interesting question.

The formal verification of static analyses, usable both to support compiler optimizations or to establish safety properties of programs, has received much attention. Considerable efforts have been expended on formally verifying Java's dataflow-based bytecode verification; see [41,56] for a survey. Cachera et al. [19] and Pichardie [82] develop a framework for abstract interpretation and dataflow analysis, formally verified using Coq. A related project is Rhodium [55], a domain-specific language to describe program analyses and transformations. From a Rhodium specification, both executable code and an automatically-verified proof of semantic preservation are generated. Rhodium achieves a high degree of automation, but applies only to the optimization phases of a compiler and not to the non-optimizing translations from one language to another.

19 Conclusions

The formal verification of a compiler back-end presented in this article provides strong evidence that the initial goal of formally verifying a realistic compiler can be achieved, within the limitations of today's proof assistants, and using only elementary semantic and algorithmic approaches. It is, however, just one exploration within a wide research area: the certification, using formal methods, of the verification tools, code generators, compilers and run-time systems that participate in the development, validation and execution of critical software. In addition, we hope that this work also contributes to renewing scientific interest in the semantic understanding of compiler technology, in

mechanized operational semantics, and in integrated environments for programming and proving.

Looking back at what was achieved, we did not completely rule out all uncertainties concerning the soundness of the compiler, but reduced the problem of trusting the whole compiler down to trusting (1) the formal semantics for the source (Cminor) and target (PPC) languages; (2) the compilation chain used to produce the executable for the compiler (Coq's extraction facility and the OCaml compiler); and (3) the Coq proof assistant itself. Concerning (3), it is true that an inconsistency in Coq's logic or a bug in Coq's implementation could theoretically invalidate all the guarantees we obtained about Compcert. As Hales [40] argues, this is extremely unlikely, and proofs mechanically checked by a proof assistant that generates proof terms are orders of magnitude more trustworthy than even carefully hand-checked mathematical proofs.

To address concern (2), ongoing work within the Compcert project studies the feasibility of formally verifying Coq's extraction mechanism and a compiler from Mini-ML (the target language for this extraction) to Cminor. Composed with the Compcert back-end, these efforts could eventually result in a trusted execution path for programs written and verified in Coq, like Compcert itself, therefore increasing confidence further through a form of bootstrapping.

The main source of uncertainty is concern (1): do the formal semantics of Cminor and PPC, along with the underlying memory model, capture the intended behaviors? One could argue that they are small enough (about 2500 lines of Coq) to permit manual review. Another effective way to increase confidence in these semantics is to use them in other formal verifications, such as the Clight to Cminor and Mini-ML to Cminor front-ends developed within the Compcert project, and the axiomatic semantics for Cminor of [3]. Future work in this direction could include connections with architectural-level hardware verification (as outlined in section 17.6) or with verifications of program provers, model checkers and static analyzers for C-like languages. Drawing and formalizing such connections would not only strengthen even further the confidence we can have in each component involved, but also progress towards the availability of high-assurance environments for the development and validation of critical software.

Acknowledgements This work was supported in part by Agence Nationale de la Recherche, grant number ANR-05-SSIA-0019. We would like to thank the following colleagues for their input. E. Ledinot made us aware of the need for a formally verified C compiler in the aerospace industry. Several of the techniques presented here were discussed and prototyped with members of the Concert INRIA coordinated research action and of the Compcert ANR advanced research project, especially Y. Bertot, S. Blazy, B. Grégoire, P. Letouzey and L. Rideau. Our formalization of dataflow analyses and constant propagation builds on that of B. Grégoire. L. Rideau and B. Serpette contributed the hairy correctness proof for parallel moves. D. Doligez proved properties of finite maps and type reconstruction for RTL. A. Appel and S. Blazy invented the continuation-based transition semantics for Cminor. G. Necula argued convincingly in favor of a posteriori verification of static analyses and optimizations. Last but not least, the author is grateful to the JAR anonymous reviewers for their meticulous reading of this long article and for their helpful suggestions for improvements. One of them went so far as to suggest the union-find based algorithm for branch tunneling presented in section 9, which is original to the best of our knowledge and more elegant than our previous bounded-depth algorithm.

References

1. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press (1998)

2. Appel, A.W.: Foundational proof-carrying code. In: Logic in Computer Science 2001, pp. 247–258. IEEE Computer Society Press (2001)
3. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007, *Lecture Notes in Computer Science*, vol. 4732, pp. 5–21. Springer (2007)
4. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Computer Aided Verification, 17th International Conference, CAV 2005, *Lecture Notes in Computer Science*, vol. 3576, pp. 291–295. Springer (2005)
5. Barthe, G., Courtieu, P., Dufay, G., Melo de Sousa, S.: Tool-Assisted Specification and Verification of the JavaCard Platform. In: Proceedings of AMAST'02, *Lecture Notes in Computer Science*, vol. 2422, pp. 41–59. Springer (2002)
6. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In: Functional and Logic Programming, 8th Int. Symp. FLOPS 2006, *Lecture Notes in Computer Science*, vol. 3945, pp. 114–129. Springer (2006)
7. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate translation for optimizing compilers. In: Static Analysis, 13th Int. Symp., SAS 2006, *Lecture Notes in Computer Science*, vol. 4134, pp. 301–317. Springer (2006)
8. Barthe, G., Kunz, C.: Certificate translation in abstract interpretation. In: Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, *Lecture Notes in Computer Science*, vol. 4960, pp. 368–382. Springer (2008)
9. Benton, N., Hur, C.K.: Biorthogonality, step-indexing and compiler correctness. In: International Conference on Functional Programming 2009, pp. 97–108. ACM Press (2009)
10. Beringer, L.: Functional elimination of phi-instructions. In: Proc. 5th Workshop on Compiler Optimization meets Compiler Verification (COCV 2006), *Electronic Notes in Theoretical Computer Science*, vol. 176(3), pp. 3–20. Elsevier (2007)
11. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer (2004)
12. Bertot, Y., Grégoire, B., Leroy, X.: A structured approach to proving compiler optimizations based on dataflow analysis. In: Types for Proofs and Programs, Workshop TYPES 2004, *Lecture Notes in Computer Science*, vol. 3839, pp. 66–81. Springer (2006)
13. Bertot, Y., Komendantsky, V.: Fixed point semantics and partial recursion in Coq. In: 10th int. conf. on Principles and Practice of Declarative Programming (PPDP 2008), pp. 89–96. ACM Press (2008)
14. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Putting it all together? Formal verification of the VAMP. Int. J. on Software Tools for Technology Transfer **8**, 411–430 (2006)
15. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods, *Lecture Notes in Computer Science*, vol. 4085, pp. 460–475. Springer (2006)
16. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning **43**(3), 263–288 (2009)
17. Blech, J.O., Glesner, S., Leitner, J., Mülling, S.: Optimizing code generation from SSA form: A comparison between two formal correctness proofs in Isabelle/HOL. In: Proc. COCV Workshop (Compiler Optimization meets Compiler Verification), *Electronic Notes in Theoretical Computer Science*, vol. 141(2), pp. 33–51. Elsevier (2005)
18. Boehm, H.J.: Threads cannot be implemented as a library. In: Programming Language Design and Implementation 2005, pp. 261–268. ACM Press (2005)
19. Cachera, D., Jensen, T.P., Pichardie, D., Rusu, V.: Extracting a data flow analyser in constructive logic. Theoretical Computer Science **342**(1), 56–78 (2005)
20. Chaitin, G.J.: Register allocation and spilling via graph coloring. In: Symposium on Compiler Construction, *SIGPLAN Notices*, vol. 17(6), pp. 98–105. ACM Press (1982)
21. Chen, J., Hawblitzel, C., Perry, F., Emmi, M., Condit, J., Coetzee, D., Pratikaki, P.: Type-preserving compilation for large-scale optimizing object-oriented compilers. In: Programming Language Design and Implementation 2008, pp. 183–192. ACM Press (2008)
22. Chirica, L., Martin, A.: Toward compiler implementation correctness proofs. ACM Transactions on Programming Languages and Systems **8**(2), 185–214 (1986)
23. Chlipala, A.J.: A certified type-preserving compiler from lambda calculus to assembly language. In: Programming Language Design and Implementation 2007, pp. 54–65. ACM Press (2007)

24. Clemmensen, G., Oest, O.: Formal specification and development of an Ada compiler. In: IEEE Conf. on Soft. Engineering, pp. 430–440. IEEE Computer Society Press (1984)
25. Coq development team: The Coq proof assistant. Software and documentation available at `http://coq.inria.fr/` (1989–2009)
26. Coupet-Grimal, S., Delobel, W.: A uniform and certified approach for two static analyses. In: Types for Proofs and Programs, Workshop TYPES 2004, *Lecture Notes in Computer Science*, vol. 3839, pp. 115–137. Springer (2006)
27. Dargaye, Z.: Vérification formelle d'un compilateur pour langages fonctionnels. Ph.D. thesis, University Paris 7 Diderot (2009)
28. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Software Engineering Notes **28**(6), 2–2 (2003)
29. Dold, A., Gaul, T., Zimmermann, W.: Mechanized verification of compiler backends. In: Int. Workshop on Software Tools for Technology Transfer, STTT 1998, *BRICS Notes*, vol. NS-98-4, pp. 13–24. University of Aarhus (1998)
30. Dold, A., Vialard, V.: A mechanically verified compiling specification for a Lisp compiler. In: Proc. FST TCS 2001, *Lecture Notes in Computer Science*, vol. 2245, pp. 144–155. Springer (2001)
31. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, pp. 255–264. ACM Press (2008)
32. Ellis, J.R.: Bulldog: a compiler for VLSI architectures. ACM Doctoral Dissertation Awards series. The MIT Press (1986)
33. Feng, X., Ni, Z., Shao, Z., Guo, Y.: An open framework for foundational proof-carrying code. In: Int. Workshop on Types in Language Design and Implementation (TLDI'07), pp. 67–78. ACM Press (2007)
34. Fox, A.C.J.: Formal specification and verification of ARM6. In: Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, *Lecture Notes in Computer Science*, vol. 2758, pp. 25–40. Springer (2003)
35. George, L., Appel, A.W.: Iterated register coalescing. ACM Transactions on Programming Languages and Systems **18**(3), 300–324 (1996)
36. Goos, G., Zimmermann, W.: Verification of compilers. In: Correct System Design, Recent Insight and Advances, *Lecture Notes in Computer Science*, vol. 1710, pp. 201–230. Springer (1999)
37. Grégoire, B.: Compilation des termes de preuves: un (nouveau) mariage entre Coq et OCaml. Ph.D. thesis, University Paris 7 Diderot (2003)
38. Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: Static Analysis, 11th Int. Symposium, SAS 2004, *Lecture Notes in Computer Science*, vol. 3148, pp. 212–227. Springer (2004)
39. Guttman, J., Monk, L., Ramsdell, J., Farmer, W., Swarup, V.: The VLISP verified Scheme system. Lisp and Symbolic Computation **8**(1-2), 33–110 (1995)
40. Hales, T.C.: Formal proof. Notices of the American Mathematical Society **55**(11), 1370–1380 (2008)
41. Hartel, P.H., Moreau, L.A.V.: Formalizing the safety of Java, the Java virtual machine and Java Card. ACM Computing Surveys **33**(4), 517–558 (2001)
42. Henderson, F.: Accurate garbage collection in an uncooperative environment. In: International Symposium on Memory Management 2002, *SIGPLAN Notices*, vol. 38(2), pp. 150–156. ACM Press (2003)
43. Hobor, A., Appel, A.W., Zappa Nardelli, F.: Oracle semantics for concurrent separation logic. In: Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, *Lecture Notes in Computer Science*, vol. 4960, pp. 353–367. Springer (2008)
44. Huang, Y., Childers, B.R., Soffa, M.L.: Catching and identifying bugs in register allocation. In: Static Analysis, 13th Int. Symp., SAS 2006, *Lecture Notes in Computer Science*, vol. 4134, pp. 281–300. Springer (2006)
45. Huet, G.P.: The Zipper. Journal of Functional Programming **7**(5), 549–554 (1997)
46. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In: Fundamental Approaches to Software Engineering, 3rd Int. Conf. FASE 2000, *Lecture Notes in Computer Science*, vol. 1783, pp. 284–303. Springer (2000)
47. IBM Corporation: The PowerPC Architecture. Morgan Kaufmann Publishers, San Francisco (1994)

48. Kildall, G.A.: A unified approach to global program optimization. In: 1st symposium Principles of Programming Languages, pp. 194–206. ACM Press (1973)
49. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Transactions on Programming Languages and Systems **28**(4), 619–695 (2006)
50. Knoop, J., Koschützki, D., Steffen, B.: Basic-block graphs: Living dinosaurs? In: Proc. Compiler Construction '98, *Lecture Notes in Computer Science*, vol. 1383, pp. 65–79. Springer (1998)
51. Knoop, J., Rüthing, O., Steffen, B.: Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Systems **16**(4), 1117–1155 (1994)
52. Lacey, D., Jones, N.D., Van Wyk, E., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. In: 29th symposium Principles of Programming Languages, pp. 283–294. ACM Press (2002)
53. Leinenbach, D., Paul, W., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: Int. Conf. on Software Engineering and Formal Methods (SEFM 2005), pp. 2–11. IEEE Computer Society Press (2005)
54. Leinenbach, D., Petrova, E.: Pervasive compiler verification. In: 3rd Int, Workshop on Systems Software Verification (SSV 2008), *Electronic Notes in Theoretical Computer Science*, vol. 217(21), pp. 23–40. Elsevier (2008)
55. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: 32nd symposium Principles of Programming Languages, pp. 364–377. ACM Press (2005)
56. Leroy, X.: Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning **30**(3–4), 235–269 (2003)
57. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, pp. 42–54. ACM Press (2006)
58. Leroy, X.: The Compcert verified compiler, software and commented proof. Available at `http://compcert.inria.fr/` (2009)
59. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008)
60. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation **207**(2), 284–304 (2009)
61. Letouzey, P.: A new extraction for Coq. In: Types for Proofs and Programs, Workshop TYPES 2002, *Lecture Notes in Computer Science*, vol. 2646, pp. 200–219. Springer (2003)
62. Letouzey, P.: Extraction in Coq: An overview. In: Logic and Theory of Algorithms, Fourth Conference on Compu tability in Europe, CiE 2008, *Lecture Notes in Computer Science*, vol. 5028, pp. 359–369. Springer (2008)
63. Li, G., Owens, S., Slind, K.: Structure of a proof-producing compiler for a subset of higher order logic. In: Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, *Lecture Notes in Computer Science*, vol. 4421, pp. 205–219. Springer (2007)
64. Li, G., Slind, K.: Compilation as rewriting in higher order logic. In: Automated Deduction, CADE-21, *Lecture Notes in Computer Science*, vol. 4603, pp. 19–34. Springer (2007)
65. Lindig, C.: Random testing of C calling conventions. In: Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, pp. 3–12. ACM Press (2005)
66. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetical expressions. In: Mathematical Aspects of Computer Science, *Proc. of Symposia in Applied Mathematics*, vol. 19, pp. 33–41. American Mathematical Society (1967)
67. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**(1), 100–107 (1998)
68. Milner, R., Weyhrauch, R.: Proving compiler correctness in a mechanized logic. In: B. Meltzer, D. Michie (eds.) Proc. 7th Annual Machine Intelligence Workshop, *Machine Intelligence*, vol. 7, pp. 51–72. Edinburgh University Press (1972)
69. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning **5**(4), 461–492 (1989)
70. Moore, J.S.: Piton: a mechanically verified assembly-language. Kluwer (1996)
71. Morrisett, G., Crary, K., Glew, N., Walker, D.: Stack-based typed assembly language. Journal of Functional Programming **12**(1), 43–88 (2002)

72. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Transactions on Programming Languages and Systems **21**(3), 528–569 (1999)
73. Muchnick, S.S.: Advanced compiler design and implementation. Morgan Kaufmann (1997)
74. Müller-Olm, M.: Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction, *Lecture Notes in Computer Science*, vol. 1283. Springer (1997)
75. Necula, G.C.: Proof-carrying code. In: 24th symposium Principles of Programming Languages, pp. 106–119. ACM Press (1997)
76. Necula, G.C.: Translation validation for an optimizing compiler. In: Programming Language Design and Implementation 2000, pp. 83–95. ACM Press (2000)
77. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Compiler Construction, 11th International Conference, CC 2002, *Lecture Notes in Computer Science*, vol. 2304, pp. 213–228. Springer (2002)
78. Necula, G.C., Rahul, S.P.: Oracle-based checking of untrusted software. In: 28th symposium Principles of Programming Languages, pp. 142–154. ACM Press (2001)
79. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science **375**(1–3), 271–307 (2007)
80. Paul, W., et al.: The Verisoft project. `http://www.verisoft.de/` (2003–2008)
81. Peyton Jones, S.L., Ramsey, N., Reig, F.: C--: a portable assembly language that supports garbage collection. In: PPDP'99: International Conference on Principles and Practice of Declarative Programming, *Lecture Notes in Computer Science*, vol. 1702, pp. 1–28. Springer (1999)
82. Pichardie, D.: Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés. Ph.D. thesis, University Rennes 1 (2005)
83. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems, TACAS '98, *Lecture Notes in Computer Science*, vol. 1384, pp. 151–166. Springer (1998)
84. Pop, S.: The SSA representation framework: semantics, analyses and GCC implementation. Ph.D. thesis, École des Mines de Paris (2006)
85. Rideau, L., Serpette, B.P., Leroy, X.: Tilting at windmills with Coq: formal verification of a compilation algorithm for parallel moves. Journal of Automated Reasoning **40**(4), 307–326 (2008)
86. Rinard, M., Marinov, D.: Credible compilation with pointers. In: Proc. FLoC Workshop on Run-Time Result Verification (1999)
87. Rival, X.: Symbolic transfer function-based approaches to certified compilation. In: 31st symposium Principles of Programming Languages, pp. 1–13. ACM Press (2004)
88. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: 15th symposium Principles of Programming Languages, pp. 12–27. ACM Press (1988)
89. Shao, Z., Trifonov, V., Saha, B., Papaspyrou, N.: A type system for certified binaries. ACM Transactions on Programming Languages and Systems **27**(1), 1–45 (2005)
90. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer (2001)
91. Strecker, M.: Formal verification of a Java compiler in Isabelle. In: Proc. Conference on Automated Deduction (CADE), *Lecture Notes in Computer Science*, vol. 2392, pp. 63–77. Springer (2002)
92. Strecker, M.: Compiler verification for C0 (intermediate report). Tech. rep., Université Paul Sabatier, Toulouse (2005)
93. Tristan, J.B., Leroy, X.: Formal verification of translation validators: A case study on instruction scheduling optimizations. In: 35th symposium Principles of Programming Languages, pp. 17–27. ACM Press (2008)
94. Tristan, J.B., Leroy, X.: Verified validation of Lazy Code Motion. In: Programming Language Design and Implementation 2009, pp. 316–326. ACM Press (2009)
95. Zuck, L.D., Pnueli, A., Fang, Y., Goldberg, B.: VOC: A translation validator for optimizing compilers. In: COCV'02, Compiler Optimization Meets Compiler Verification, *Electronic Notes in Theoretical Computer Science*, vol. 65(2), pp. 2–18. Elsevier (2002)